DepState: Detecting Synchronization Failure Bugs in Distributed Database Management Systems

CUNDI FANG*, National University of Defense Technology, China
JIE LIANG*, Beihang University, China
ZHIYONG WU, Tsinghua University, China
JINGZHOU FU, Tsinghua University, China
ZHOUYANG JIA, National University of Defense Technology, China
CHUN HUANG, National University of Defense Technology, China
YU JIANG†, Tsinghua University, China
SHANSHAN LI†, National University of Defense Technology, China

Distributed Database Management Systems (DDBMSs) are crucial for managing large-scale distributed data. Unlike single-node databases, they are deployed across clusters, distributing data among multiple nodes. The synchronization process in DDBMSs maintains data consistency against data and cluster updates. Due to its complexity, synchronization bugs are inevitable and may cause data inconsistencies, transaction errors, or cluster crashes, severely compromising the availability and reliability of a DDBMS. However, there has been relatively little focus on testing the DDBMS synchronization process.

In this paper, we propose DepState, a framework to detect synchronization failure bugs. DepState enhances synchronization testing by simulating the complexities of data sharding and dynamic cluster conditions. It establishes dependencies between tables across nodes and systematically introduces controlled variations in cluster states. We utilize DepState on four DDBMSs: MySQL NDB Cluster, MySQL InnoDB Cluster, MariaDB Galera Cluster, and TiDB Cluster, discovering 25 new bugs, with 13 confirmed. We compare DepState against state-of-the-art tools. DepState finds 14 more synchronization failure bugs and covers 6.13%-66.51%, 5.82%-57.28%, 14.12%-83.30%, 36.81%-83.88%, and 43.24%-54.28% more lines in synchronization-related functions than Jepsen, Mallory, SQLsmith, SQLancer, and Mozi in 24 hours, respectively.

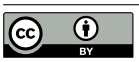
CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Information systems → Relational parallel and distributed DBMSs**; Distributed data locking.

Additional Key Words and Phrases: Distributed Database Management Systems; Synchronization Failure

ACM Reference Format:
Cundi Fang, Jie Liang, Zhiyong Wu, Jingzhou Fu, Zhouyang Jia, Chun Huang, Yu Jiang, and Shanshan Li. 2025. DepState: Detecting Synchronization Failure Bugs in Distributed Database Management Systems. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA088 (July 2025), 22 pages. https://doi.org/10.1145/3728965

*Cundi Fang and Jie Liang contributed equally to this work.
†Yu Jiang and Shanshan Li are the corresponding authors.

Authors' Contact Information: Cundi Fang, National University of Defense Technology, College of Computer Science and Technology, Changsha, China, fangcundi@nudt.edu.cn; Jie Liang, Beihang University, Beijing, China, liangjie.mailbox.cn@gmail.com; Zhiyong Wu, Tsinghua University, Beijing, China, 253540651@qq.com; Jingzhou Fu, Tsinghua University, Beijing, China, fuboat@outlook.com; Zhouyang Jia, National University of Defense Technology, College of Computer Science and Technology, Changsha, China, jiazhouyang@nudt.edu.cn; Chun Huang, National University of Defense Technology, College of Computer Science and Technology, Changsha, China, chunhuang@nudt.edu.cn; Yu Jiang, Tsinghua University, Beijing, China, jiangyu198964@126.com; Shanshan Li, National University of Defense Technology, College of Computer Science and Technology, Changsha, China, shanshanli@nudt.edu.cn.



ACM 2994-970X/2025/7-ARTISSTA088
https://doi.org/10.1145/3728965

1 Introduction

Distributed Database Management Systems (DDBMSs) are extensively employed in large-scale data processing and high-concurrency scenarios, such as data warehousing, real-time analytics, e-commerce platforms [35]. As organizations increasingly rely on DDBMSs to handle massive datasets and high-concurrency workloads, ensuring the synchronization of data updates has become critical [7]. Maintaining data consistency during updates of shared data and changes in node clusters is crucial for ensuring system performance and reliability. The synchronization process coordinates data updates and transaction processing across nodes, ensuring data consistency, not only the consistency of data replicas, but also the consistency of data and metadata between nodes when executing transactions, thereby enhancing system stability, fault tolerance, and scalability.

Synchronization in DDBMSs is complex, requiring consistency and reliability across nodes during transactions. The complexity stems from data interdependencies, as shared data across nodes requires strict synchronization, where delays or failures on one node can compromise global consistency [38]. Therefore, the system must have strong coordination and fault-tolerance capabilities. Moreover, the complexity of clusters is reflected in their *frequent state changes*. Node additions, removals, or faults introduce uncertainty into the synchronization process [21], requiring the system to adjust synchronization strategies to ensure correct transaction execution.

As a result, avoiding implementation errors in synchronization processes proves to be challenging. Given that the synchronization process is critical to preserving the consistency properties of a DDBMS [36], errors in the synchronization process can lead to failures. Such failures can trigger severe consequences, such as service outages or cluster crashes, which significantly impact system availability. For example, in 2018, GitHub experienced a 24-hour service disruption due to a bug in the synchronization implementation, which was triggered by a 43-second network disconnect that affected the connection between webhooks and its MySQL database [39]. Similarly, in 2020, a bug in the synchronization mechanism of Google Cloud Spanner disrupted essential services like Gmail and YouTube, leading to significant outages for users worldwide and financial losses for businesses relying on these platforms [16]. Such bugs in the synchronization mechanism that cause the synchronization process to fail, leaving the system in an inconsistent or unavailable state, are what we refer to as **Synchronization Failure Bugs**. These bugs can disrupt distributed systems and compromise data integrity, leading to significant operational issues.

Numerous studies have been proposed to detect bugs in DDBMSs. For example, Jepsen [17] uses the causal consistency detector Elle [18] and fault generators to simulate network partitions, aiding in the identification of errors in DDBMSs. Similarly, Mallory [26] enhances Jepsen by optimizing the fault injection process with fuzzy processing of the Lamport graph, facilitating a more efficient exploration of the fault space. However, most of them focus on fault tolerance and recovery mechanisms, which often overlook synchronization failure bugs in DDBMSs. Unlike general distributed systems, DDBMSs must ensure complex database transaction attributes (like ACID) while ensuring data consistency across multiple nodes. Therefore, DDBMS bugs are often triggered under databases with complex data dependencies or changes in cluster nodes, which makes it hard for other tools to detect and resolve these issues effectively.

To identify *synchronization failure bugs* in DDBMS, we face two challenges: **(1) Establish complex dependencies across sharded data tables.** In DDBMSs, data is interconnected and dynamic. Dependencies vary between partitions and evolve. To identify bugs, it is essential to design a data schema that captures these diverse dependencies while preserving semantic relationships across nodes. **(2) Explore cluster state change sequences aligned with synchronization phases.** Cluster state change sequences are the various states of nodes in a DDBMS over time. Capturing the timing and context of cluster state changes during specific synchronization phases

is difficult, as these changes must align with dynamic conditions within the cluster. Moreover, the expansive state space complicates generating sequences that could trigger issues, as many synchronization failures only appear under specific timing and execution conditions.

In order to address these challenges, DepState is introduced as a fuzzing framework specifically engineered to identify synchronization failure issues in DDBMSs. DepState first utilizes **dependency-aware sharding data generation** to create diverse scenarios that mimic real-world conditions. It analyzes table relationships to generate foreign key links across shards, ensuring that changes in one table dynamically impact related tables. Furthermore, DepState integrates data integrity constraints to ensure the precise modeling of real-world scenarios. Moreover, we propose **synchronization-sensitive cluster state sequence exploration** to precisely modify cluster states during different synchronization phases. This methodology investigates novel test sequences that emerge at distinct synchronization intervals via mutation, leveraging coverage feedback to efficiently identify potential synchronization issues.

We implement DepState and evaluate it on four widely-used DDBMSs: MySQL NDB Cluster, MySQL InnoDB Cluster, MariaDB Galera Cluster, and TiDB Cluster. DepState reports a total of 25 *synchronization failure bugs*, with 13 anomalies have been confirmed. In addition, we compare DepState with the state-of-the-art distributed system testing tool Jepsen and Mallory, and advanced database testing tools SQLsmith, SQLancer, and Mozi. The 24-hour result shows that DepState found 14 more synchronization failure bugs, and covered 6.13%-66.51%, 5.82%-57.28%, 14.12%-83.30%, 36.81%-83.88%, and 43.24%-54.28% more lines in synchronization-related functions than Jepsen, Mallory, SQLsmith, SQLancer, and Mozi, respectively.

In summary, we make the following contributions:

- We identified that synchronization process in DDBMSs can be faulty and lead to serious issues such as service interruptions, but state-of-the-art testing techniques pay little attention to the synchronization failure bugs.
- We designed and implemented a fuzzing framework DepState, which combines dependency-aware sharding data generation and synchronization-sensitive cluster state sequence exploration to detect synchronization failure bugs.
- We demonstrated that DepState improves synchronization code coverage compared to existing state-of-the-art tools. Additionally, we tested four widely-used DDBMSs and uncovered 25 new *synchronization failure bugs*.

2 Background and Motivation

2.1 DDBMS Overview

A Distributed Database Management System (DDBMS) fundamentally extends a single-node database to address the scalability challenges inherent in large-scale data management. In traditional single-node DBMSs, all data is stored and managed centrally, which simplifies transaction processing and synchronization mechanisms. However, as data volumes exponentially grow, this centralized architecture proves inadequate for efficiently managing and processing massive datasets.

To enhance scalability, DDBMSs implement data sharding, partitioning databases into smaller shards stored on separate nodes, enabling parallel processing. This reduces node load and improves throughput and response times. Systems like Google Spanner[6] and MongoDB[3] demonstrate the adoption of sharding in large-scale environments. A cluster can remain operational by leveraging load balancing and redundancy even if one or more nodes fail.

Once data is distributed across shards and clusters, the synchronization process becomes pivotal for coordinating updates and managing concurrency. DDBMS must orchestrate transaction processing and synchronization operations among multiple nodes, making its design more complex than

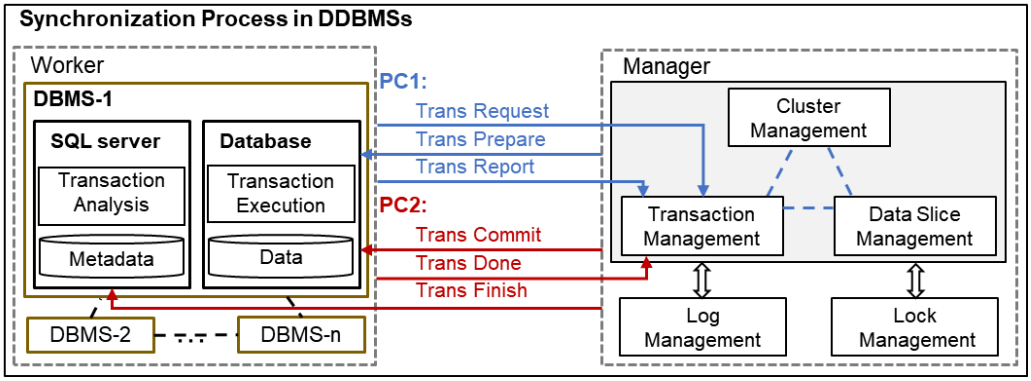


Fig. 1. A Typical Synchronization Process in DDBMSs That Use the 2PC Protocol.

single-node systems. In particular, cross-shard data dependencies and dynamic cluster states must be managed simultaneously to ensure consistency and correctness. Consequently, synchronization protocols in DDBMS often evolve from a traditional two-phase commit[11] or employ more advanced consensus algorithms (e.g., Paxos[20] or Raft[27]) to handle distributed coordination.

2.2 Synchronization Process in DDBMS

The **synchronization process** ensures data consistency and reliability across nodes in a DDBMS, which maintains ACID properties [12] while addressing high availability and concurrency. This mechanism adapts traditional database characteristics.

Figure 1 shows the synchronization process in DDBMS using the 2PC protocol [11]. Upon receiving a transaction, the SQL server forwards it to the Transaction Manager, which assigns it to the relevant node. In phase one, the node prepares the transaction and reports the outcome. In phase two, the manager aggregates results and decides on committing or rolling back the transaction. If the transaction affects metadata, consistency across nodes and slices is ensured. Additionally, modules are integrated to manage cluster state changes, such as node additions or removals.

While maintaining the features of traditional databases, the synchronization mechanism of DDBMS must also tackle the challenges posed by *data sharding* and *cluster state changes* [30]. Unlike traditional DBMS, DDBMS must manage relationships between data stored across multiple nodes, requiring coordination of both intra-table dependencies and cross-node interactions. Maintaining dependency consistency across all nodes ensures that even localized updates must reflect across the entire system. This distributed architecture amplifies the complexity of synchronization [34].

Table 1. Some Cluster Operations and Their Impacts on the Synchronization Process

Cluster Operation	Impact on Synchronization
Add Data Node	Triggers load balancing and data redistribution.
Remove Data Node	Affects dependencies and synchronization.
Data Backup	Requires synchronization across all nodes.
Cluster Restart	Synchronizes all nodes during startup.
Data Node Restart	Involves independent synchronization.
Forced Data Synchronization	Forces synchronization by user intervention.

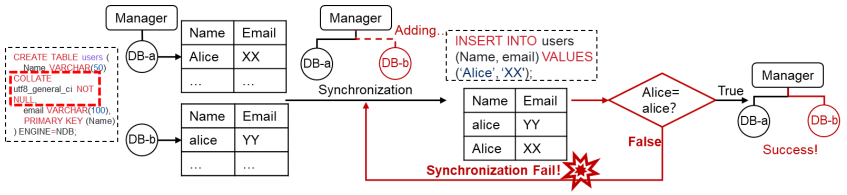


Fig. 2. Bug #98526 in MySQL NDB Cluster: Forced Shutdown When a Data Node Joins the Cluster.

Cluster state changes, such as node additions, complicate DDBMS synchronization, particularly regarding consistency and availability [19]. For instance, node restarts during transaction commits can leave transactions in uncertain states, destabilizing subsequent operations. Cluster state modifications occur through **cluster operations**, which include adding, removing, or reconfiguring nodes, and adjusting load distribution, resource allocation, and data synchronization. Table 1 outlines these operations and their synchronization impacts. Such changes can trigger complex synchronization behaviors. Moreover, frequent state transitions complicate distributed lock management, potentially causing deadlocks or resource contention if locks are not properly released.

2.3 An Example of Synchronization Failure Bug

We use a synchronization failure bug in MySQL NDB Cluster (Bug #98526) [15] as a representative example to highlight the symptoms of such bugs, the substantial risks their to system integrity, root causes, and the complexities of accurately identifying them. Specifically, this bug resulted in a catastrophic cluster crash during the attempt to add a data node into the cluster, thereby illustrating how synchronization issues can induce significant system instability and operational disruptions.

2.3.1 Bug Description and Root Cause. The bug, triggered during primary key comparison post-node connection in the synchronization process of MySQL NDB Cluster, violated cross-node primary key constraints, leading to synchronization failures, node join issues, and a crash of Node B's service. It was resolved within three days and documented in MySQL Cluster's release notes.

Figure 2 illustrates the trigger process of this bug. As it shows, the table `users` has a primary key constraint on the `Name` column, which enforces case-insensitive matching. This constraint prevents entries such as "Alice" and "alice" from coexisting in the same column. In this scenario, Node A, already part of the cluster, contains a record with Name='Alice'. Node B, which is not yet part of the cluster, holds a record with Name='alice'. During the process of adding node B to the cluster, "Alice" and "alice" should be considered equal, and one version of the record should be selected as authoritative to maintain the integrity of the primary key constraint. However, the binary representations of "Alice" and "alice" are mistakenly treated as distinct, causing both records to coexist on Node B and violating the primary key constraint.

The bug's root cause stems from an implementation error: the primitive binary comparison logic fails to handle character-based primary keys with case-insensitive collation rules, resulting in a synchronization failure. The developer addressed this issue by modifying the function `handle_nr_copy` in the file`DblqhMain.cpp`. This patch introduced a conversion mechanism for character-type primary keys, which includes an additional character set conversion step to ensure that logically equivalent primary keys are correctly matched, even if their binary representations differ. To trigger this issue, two basic conditions are needed: first, conditional constraints between data on different nodes, and second, changes in the cluster activate the synchronization process.

2.3.2 Limitation of Existing Methods to Find This Bug. Current DDBMS testing methods fail to detect this bug due to inability to manage data dependencies and dynamic cluster behaviors. For

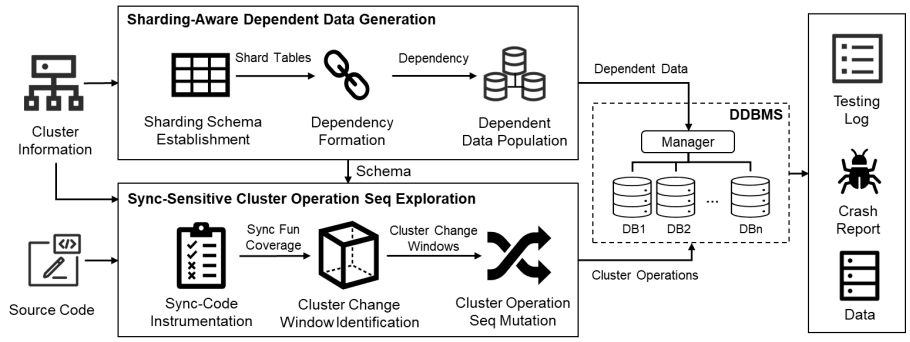


Fig. 3. Overview of DepState. It generates a database with interdependent tables distributed across nodes and employs checkpointing to capture synchronization phases, pinpointing specific state changes. Using coverage feedback, DepState then explores new test sequences through mutation to detect potential issues.

instance, tools like Jepsen and Mallory trigger faults via fault injection but overlook data complexities. They are insufficient for fine-grained dependencies, like foreign and primary keys affecting synchronization. Thus, Figure 2 shows primary key constraints beyond the tools' capabilities.

Moreover, existing methods fail to address the complexity of the synchronization phase when the cluster state changes frequently. Normal cluster changes, such as node connections and restarts, cluster restarts, can disrupt the synchronization process, particularly during the commit phase, leading to transaction failures or consistency interruptions. Advanced tools like Jepsen and Mallory primarily focus on fault injection and do not explore these interactions in depth. As a result, these approaches are limited in detecting the impact of normal cluster state changes on synchronization.

3 Design of DepState

Figure 3 presents the structure of DepState, which includes two core components: sharding-aware data generation and synchronization-sensitive state exploration. (1) The first module generates diverse test data and analyzes its storage across partitions, factoring in the current sharding state. By linking data within a shard to specific items in other shards (e.g., primary and foreign keys), it increases synchronization complexity to mimic real-world issues. (2) The second module captures synchronization phases through a checkpoint mechanism to identify critical moments for state alterations. Next, it implements state operations that control changes at each node, ensuring that modifications align with the ongoing synchronization context. Finally, it explores novel test sequences occurring during specific synchronization timings, utilizing mutation and coverage feedback to uncover potential synchronization issues effectively.

3.1 Sharding-Aware Dependent Data Generation

To tackle the challenges of establishing complex dependencies between tables across shards in DDBMS, we design sharding-aware dependent data generation. It constructs a distributed schema that incorporates sharding and dependencies, generating compliant data for the synchronization process. Figure 4 shows the overall process of sharding-aware dependent data generation.

① First, DepState systematically constructs a distributed schema that encompasses both sharding parameters (e.g., the number of shards) and the partitioning strategy (e.g., hash partitioning). The two CREATE TABLE statements in the figure define tables T1 and T2, with 1 shard and 2 shards respectively, while also specifying their data types, primary keys, and structural relationships.

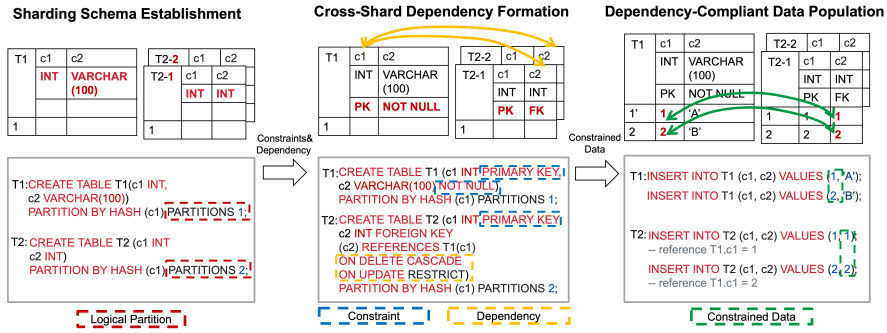


Fig. 4. An Example of Sharding-Aware Dependent Data Generation. DepState begins by specifying the column types and partitioning scheme for the table, then proceeds to add constraints and inter-table dependencies, and finally generates data that adheres to the defined constraints and dependencies.

② Second, DepState generates intra-table constraints and inter-table dependencies across partitions, thereby augmenting the complexity of the distributed database schema. The SQL statements within the blue box in the figure represent the addition of foreign key constraints, while the yellow box SQL statements introduce dependency operations, establishing a foreign key relationship between column c1 of T1 and column c2 of T2.

③ Finally, DepState generates dependency-compliant data to populate the database, creating scenarios that reflect the nuanced, shard-partitioned distribution of tables across nodes, relecting the complexities inherent in DDBMS architectures. The generated INSERT statements in the figure, highlighted in green, represent constraint data created due to inter-table dependencies.

3.1.1 Sharding Schema Establishment. Most DDBMSs autonomously manage data sharding and distribution, restricting direct user control over sharding boundaries [1]. However, users can influence data layout by designing distributed sharding schemes that involve partition keys and configurations. DDBMSs provide configurable options for table partitioning (e.g., the PARTITION command in MySQL NDB Cluster), allowing customization of key elements such as sharding keys, types (range, list, hash, or composite), shard counts, and maintenance strategies.

Based on that, we categorize database tables into two types: **Centralized Sharding Tables** and **Decentralized Sharding Tables**. Centralized sharding means that data tables have fewer shards, which are more concentrated on one or a few data nodes. In contrast, decentralized sharding refers to data tables having a larger number of shards, which are more evenly distributed across a greater number of data nodes. A subset of tables is randomly designated as Centralized Sharding Tables, which have a limited number of shards. These tables are marked in the schema to record their partition count. The remaining tables are classified as Decentralized Sharding Tables, which are configured with a greater number of logical partitions and similarly documented. For example, as shown in Figure 4, T1 and T2 are two tables that are sharded using a hash sharding strategy.Among them, "create table" indicates that this is a statement to build a table, "int" and "varchar(100)" specify the attributes of each column in the table, and "partition by hash" is used to enforce the number of slices. T1 is classified as a Centralized Sharding Table with one shard, while T2 is categorized as a Decentralized Sharding Table with two shards (i.e., T2-1 and T2-2).

When establishing dependencies between tables, we link Centralized and Decentralized Sharding Tables. Such links add complexity to synchronization, as they demand careful coordination across shards to maintain data consistency and integrity. The absolute difference in shard counts serves as a key indicator for generating these dependencies.

In distributed databases, the data load balancing mechanism operates transparently, thereby restricting the ability to specify specific storage nodes. During the establishment of dependencies between tables, scenarios where one table is confined to a limited number of nodes while the other is spread across a greater number of nodes tend to foster inter-node dependencies. This results in more intricate synchronization scenarios, thereby elevating the risk of errors. Consequently, our objective is to maximize the disparity in the number of partitions between the two tables.

3.1.2 Cross-Shard Dependency Formation. In DDBMSs, foreign key dependencies are typically established on primary keys or unique constraints. For any given table, we designate the first column as the primary key for sharding, and its type and other attributes will not be modified thereafter. For the other columns, the constraint can be any one of $\{NOTNULL, UNIQUE, CHECK\}$. The first column is chosen as the primary key because all columns in a database table are equivalent, and foreign key dependencies require the original column to have primary or unique key constraints. To streamline the standardization of our inter-table dependency generation component, we selected the first column as the primary key, simplifying SQL statement generation. Foreign key dependencies create relationships across tables and shards, but in distributed environments, two main challenges arise: ensuring integrity across nodes and generating dependencies that span multiple shards.

To maintain the integrity of foreign key constraints, the following conditions must be met: (1) Data type alignment: The data types of the foreign key column and the referenced column must match to prevent type mismatches. (2) The referenced column must be either a primary key or a unique column to uphold referential integrity. Algorithm 1 shows the steps of establishing foreign key dependencies: ① Pair the eligible tables, sorting them by the absolute difference in partition counts in descending order, and by table index in descending order if the partition differences are equal (Lines 2 to 7). This step lays the groundwork for more robust construction of dependencies between nodes and the creation of more complex data scenarios. ② Select the table pair with the

Algorithm 1: Foreign Key Dependency Establishment

Input : T: List of tables with schema information, P: Partition count array for each table, C: Column data types and uniqueness constraints
Output : T with foreign key constraints

1 Initialize $pairs \leftarrow \emptyset$, $processed \leftarrow \emptyset$;
2 **for** $i, j \in \{0, \ldots, n-1\}, i < j$ **do**
3 **if** $\exists c_i \in T_i, c_j \in T_j$ *s.t.* $Match(c_i, c_j)$ **then**
4 $pairs \leftarrow pairs \cup \{(i, j, |P[i] - P[j]|)\}$;// Store table pairs with partition difference
5 **end**
6 **end**
7 Sort $pairs$ by $|P[i] - P[j]|$ in descending order;
8 **foreach** (A, B) *in pairs* **do**
9 **if** $A, B \notin processed$ **then**
10 $(c_A, c_B) \leftarrow$ find_match(T_A, T_B);// Find columns matching Foreign Key conditions
11 $fk\_sql \leftarrow$ generate_FK_sql(c_A, T_B, c_B);// Generate Foreign Key SQL with random actions
12 Insert fk_sql into T_A;
13 $processed \leftarrow processed \cup \{A, B\}$;
14 **end**
15 **end**
16 **return** T;

largest absolute difference in partition count to establish the foreign key relationship (Line 8). ③ Identify the primary key and foreign key columns, ensuring that the foreign key column meets data type consistency and uniqueness constraints (Line 10). ④ Establish the foreign key constraint to ensure that the foreign key column effectively references the primary key column. Tables with established foreign keys will not be processed further (Lines 12-13). ⑤ Repeat the steps until all qualifying tables have established their corresponding foreign key dependencies (Line 8-15).

For instance, In $T2$, column c_2 is a foreign key referencing $T1.column1$. We first check if $T2.column2$ and $T1.column1$ have matching data types, which they do (both are INT). Then, we confirm that $T1.column1$ satisfies the uniqueness constraint as a primary key. With both conditions satisfied, the foreign key constraint is applied to $T2.column2$, as shown in the yellow box of Figure 4.

3.1.3 Dependency-Compliant Data Population. The final step entails generating data that rigorously adheres to the schema, with particular attention to the intricate constraints and complex interdependencies across shards. Algorithm 2 meticulously delineates the data generation procedure. This algorithm systematically generates data based on dependency levels (i.e., the number of foreign key relationships associated with each table), progressing from the lowest to the highest.

Specifically, for each column exhibiting a foreign key dependency, the algorithm recursively identifies the root reference of the dependency (Line 6). It subsequently determines the permissible data range and probabilistically selects values within this range to ensure the integrity and consistency of the data (Line 8). This approach ensures that all insertion operations strictly adhere to complex column constraints and cross-partition dependencies, thereby constructing a distributed database with sophisticated and intricate constraints.

3.2 Synchronization Sensitive Cluster Operation Sequence Exploration

To address cluster operation exploration in synchronization phases, we propose *synchronization-sensitive cluster operation sequence exploration* to trigger synchronization failure bugs by precisely capturing these phases and managing DDBMS cluster changes.

Algorithm 2: Data Generation with Constraints

Input : S: Table schema with constraints,
D: Inter-table dependency map,
P: Partition and sharding info
Output : DB: Populated distributed database

1 Initialize data pool $V \leftarrow \emptyset$;
2 Sort tables T_i by dependency depth;
3 **foreach** T_i *in ascending depth* **do**
4 Initialize $R_{T_i} \leftarrow \emptyset$ (row list for T_i);
5 **foreach** *column* $c_j \in T_i$ *with foreign key* **do**
6 $r_j \leftarrow$ find_root_dependency(c_j, D);// Find the root dependency
7 Range$(r_j) \leftarrow$ data_range(r_j, P);
8 $V[c_j] \leftarrow$ random_select(Range(r_j));
9 **end**
10 Generate rows for T_i using $V[c_j]$ and append to R_{T_i};
11 Insert R_{T_i} into T_i;
12 **end**
13 **return** DB;

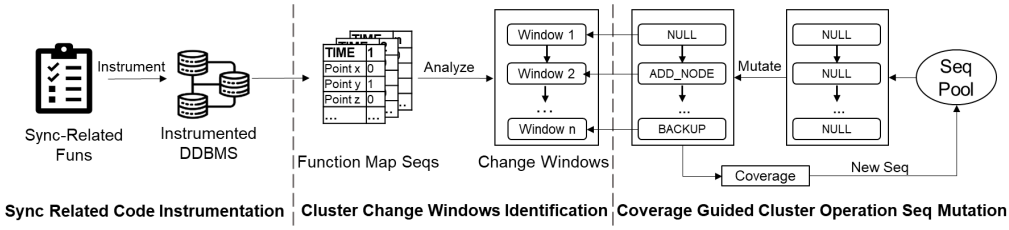


Fig. 5. The Process of Synchronization-Sensitive Cluster Operation Sequence Exploration. First, DepState performs synchronization-related instrumentation to gather runtime data. Based on this information, DepState then identifies synchronization-sensitive points and, finally, generates a Cluster Operation Sequence.

We use **Cluster Operation Sequences** to describe a series of actions performed on a cluster. To explore various scenarios in the synchronization process as thoroughly as possible, we perform multiple cluster operations instead of just one, creating Cluster Operation Sequences. To effectively influence the synchronization process through cluster state changes, two fundamental issues need to be addressed: firstly, *when to perform each cluster operation in the sequence*, and secondly, *determining each cluster operation in the sequence*. To address the first issue, cluster state changes can be made heuristically during periods of high interactivity among cluster nodes, such as during resource operations. We call these periods **Cluster Change Windows**. Different SQL sequences may lead to varying Cluster Change Windows due to transaction complexity and workload patterns. Cluster changes during this period may complicate synchronization and increase bug occurrence. Coverage can guide the exploration of the Cluster Operation Sequence, enabling systematic state space analysis and ensuring comprehensive system performance evaluation.

Figure 5 shows the process of synchronization-sensitive cluster state sequence exploration. ① Firstly, DepState instruments synchronization-related code to gather synchronization phase messages. ② Secondly, DepState generates and executes a workload while monitoring synchronization behaviors, collecting time-series data for various synchronization phases. For example, when an *ALTER* statement is executed, DepState tracks the synchronization process between the management node and other nodes, segmenting the process based on the management node's wait behavior to generate a time series. ③ Thirdly, DepState explores the Cluster Operation Sequence using coverage feedback. Continuing with the previous example, DepState inputs the previous time series and randomly generates cluster state modification behaviors in the blank windows, outputting a time series containing state modification operations. Then DepState executes the same *ALTER* sequence, dynamically monitoring its execution. Based on the output time series, DepState executes specific modification behaviors at designated synchronization stages, detects failures, and measures coverage. When coverage plateaus, DepState restores the cluster to its initial state and repeats the process for the next iteration. We present a detailed example in repository.

3.2.1 Synchronization Related Code Instrumentation. To comprehensively capture the various phases of the synchronization process and identify critical synchronization points, DepState first instruments the DDBMS source code to collect execution logs.

Specifically, DepState instruments the *synchronization-related functions* to capture runtime messages associated with the synchronization process. These functions are designed to handle various aspects of the synchronization process, such as initiation, inter-node communication, and data synchronization. For each synchronization-related function, DepState inserts *"logging code"*, which is intended to log execution paths and capture real-time messages throughout the synchronization process. We developed an automated script that uses keywords to identify matching functions

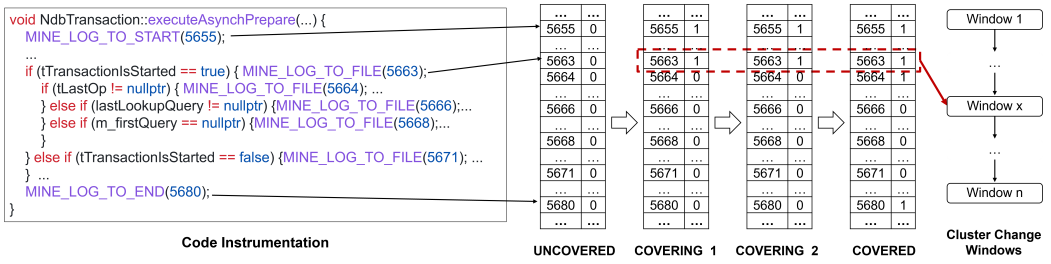


Fig. 6. Example of the Cluster State Change Behavior Sequence Generator.

and instrument their entry, exit, and conditional branches with logging code. The keyword-based approach was chosen as synchronization-related code typically exhibits similar naming patterns, resulting in fewer false negatives. For example, the three synchronization-related functions in NDB, `execSYNC_PAGE_CACHE_CONF`, `execSYNC_EXTENT_PAGES_CONF`, and `execSYNC_PAGE_CACHE_REQ`, all share common keywords. About reducing false positives, we found that synchronization-related code is usually located within the same directory, for example, these three functions are all found in the *storage/ndb/src/kernel/blocks/* path in NDB. Our tool thus supports a specified search range to reduce false positives. In our experiments, we identified the paths to the synchronisation-related code and the keywords for the NDB cluster through half an hour of code analysis. All synchronization-related functions are mapped to a set comprising `UNCOVERED`, `COVERING`, and `COVERED`. Before the initiation of the DDBMS synchronization process, all synchronization-related functions are categorized as `UNCOVERED`, indicating that they remain unexecuted. During the synchronization process, the "logging code" continuously updates the status of each function in real time.

For example, the left part of Figure 6 shows a segment of instrumentation code. In the function `NdbTraction::executeAsynchPrepare`, DepState inserts a start-point (i.e., `MINE_FUNCTION_START`) and an end-point (i.e., `MINE_FUNCTION_END`) to systematically track whether the function is covered. The tool identifies and associates each function's exit points, enabling precise instrumentation. Specifically, if the start-point remains uncovered, the "logging code" indicates that the function has not been executed (i.e., `UNCOVERED`). If the start-point is covered while the end-point remains uncovered, the "logging code" indicates that the function is actively in execution (i.e., `COVERING`). If both points are covered, the "logging code" will indicate the function has been successfully executed (i.e., `COVERED`). In addition, DepState instruments the function by adding the `MINE_LOG_TO_FILE` code to record the covered branches in this function. If the `MINE_LOG_TO_FILE` code is triggered, it indicates that the corresponding branches are covered.

3.2.2 Cluster Change Windows Identification. DepState first generates and executes a set of SQL statements to trigger the synchronization process and identifies Cluster Change Windows. The set of statements includes not only SQL's DQL (Data Query Language) but also DDL (Data Definition Language) and DML (Data Manipulation Language). The introduction of DDL may lead to changes in metadata, which could potentially trigger more complex synchronization operations. However, in order to prevent any impact on the existing schema, the execution of a DDL statement is immediately followed by a statement that resets the metadata to its previous state.

DepState identifies Cluster Operation Windows by determining the precise activity status of the synchronization function. For instance, within a typical two-phase protocol employed in the synchronization process, the management node initially analyzes the transaction and systematically distributes its relevant information to the respective data nodes during the preparation phase. Subsequently, each data node engages in a series of intricate interactions, including local checks,

resource locking, and pre-execution procedures. The occurrence of cluster changes during this phase significantly escalates the complexity and intricacy of synchronization. Since the management node awaits feedback from each data node, its synchronization function remains temporarily inactive.

In the execution phase, the management node dispatches execution instructions upon confirming the readiness of each data node. Subsequently, each data node executes the actual commit operation and returns the result to the management node upon confirmation. Making cluster state changes during this period also triggers a complex synchronization process, and the management node again waits for operations from the data nodes, leaving its synchronization function inactive. This represents the second Cluster Operation Window. If a data node fails to execute the commit, it will send a rollback request to the management node. The management node will confirm this and issue a rollback command to each data node. At this stage, the management node will once again enter a waiting state, signifying the third potential Cluster Operation Window.

The synchronization function of the management node is usually inactive during the Cluster Operation Window. This assumption is based on our observations of synchronization mechanisms within Distributed Database Management Systems (DDBMS). For instance, in a two-phase synchronization mechanism, the synchronization process necessitates communication and interaction between the management node and other data nodes to determine whether and how to proceed with the next synchronization step. During this period, the management node awaits the completion of internal operations by the other nodes and the return of their results. Due to this busy-waiting state, the synchronization function of the management node typically remains inactive until either the results are received or a predefined waiting time threshold is reached. This observation underpins the hypothesis presented in our paper. Furthermore, upon analyzing synchronization mechanisms in other DDBMSs, such as three-phase synchronization[8] and group replication[40].

Consequently, regardless of the employed synchronization scheme, as long as the management node is required to wait for feedback from other nodes during synchronization, our proposed method can effectively partition the entire process into clearly identifiable phases. Thus, extended periods during which this function's state remains completely unchanged can be reliably detected. Specifically, this window is heuristically identified by examining the *"logging code"*; if the synchronization function remains unchanged for a period, a Cluster Operation Window is signaled. The right part of Figure 6 illustrates how DepState identifies the Cluster Change Window. If `NdbTransaction::executeAsynchPrepare` function in the manager code executes, DepState updates its status to `COVERING`. If it remains in this state and its branch coverage does not change for a significant period, it indicates a potential Cluster Operation Window.

3.2.3 Coverage Guided Cluster Operation Sequence Mutation. For each iteration of the set of SQL expressions executed, we get a series of clustered operation windows. The synchronization operations we perform in each window constitute the sequence of synchronization operations.

In order to induce a greater variety of behaviors during the synchronization process, DepState generates a sequence of cluster operations based on the feedback obtained from transition-operations coverage. Transition-operations coverage records the sequences of cluster operations employed to modify the DDBMS cluster state, thereby facilitating the exploration of additional behaviors within the synchronization process. Algorithm 3 shows the process of state transition operations generation. Firstly, DepState maintains a sequence pool comprising sequences of cluster operations that have been employed to alter the cluster state. Upon the commencement of testing, DepState continuously mutates sequences of state transition operations to change the cluster state in a loop. In each iteration, the algorithm randomly selects a sequence from the sequence pool and identifies a specific phase within that sequence to serve as the injection point (Line 3). Next, a feasible cluster state change operation, applicable to the current cluster state is chosen from

Algorithm 3: Synchronously Guided Cluster State Changes Sequence Mutations

Input : P : Predefined-set initialized sequences,
O : Available operations collection
Output : L : Detected failures log

1 Initialize failure log $L \leftarrow \emptyset$;
2 **while** *Coverage metric Cov is increasing* ***or*** *time limit not reached* **do**
3 $S \leftarrow$ rand_time_sequence(P); // Randomly sampled time sequence
4 $t \leftarrow$ rand_time_point(S); // Randomly sampled time points
5 $o_{avail} \leftarrow$ applicable_operation_choose(O); // Choose applicable operation
6 **if** *time_point_checker*(t) **then**
// Check whether the behavior was inserted at that time
7 Insert operation o_{avail} at time t in sequence S;
8 **else**
9 Replace the operation at t with o_{avail} in S;
10 **end**
11 **if** *Execute*(S) $\rightarrow$ *Failure* **then**
12 $L \leftarrow L \cup \{Failure, S\}$;
13 **end**
14 **if** *update_coverage*(S) **then**
15 $P \leftarrow P \cup \{S\}$; // Add updated sequence to pool if coverage improves
16 **end**
17 **end**
18 **return** L;

the operation set O (Line 4). The mutation is applied by either inserting the selected operation at position t or replacing the existing operation at that position in the sequence (Lines 5–9). After the mutation, the modified test sequence is executed, and the system records whether a failure occurs (Line 10) and whether code coverage has increased (Line 13). If coverage is elevated, the mutated sequence is reintroduced to the sequence pool, permitting its consideration in subsequent testing iterations (Line 14). Eventually, the failure log and the updated sequence pool are returned.

4 Implementation

We implemented DepState using 8,843 lines of C++ code and 688 lines of Python code. Building upon SQLsmith, we developed test functionalities for the synchronization process of DDBMSs through data generation and sequence generation.

DepState consists of two main components: the sharded data generator and the cluster state sequence generator. The sharded data generator implements Algorithms 1 and 2 to build distributed database schemas and create dependencies between tables across shards. The sequence generator produces cluster operation sequences by implementing Algorithm 3, which utilizes synchronization phase information obtained from staking to simulate realistic cluster behaviors.

Adapting DepState to a new DDBMS involves modifying the client to send SQL commands that match those of the target DDBMS, thereby simulating concurrency scenarios. Additionally, the client must be customized to execute corresponding commands that alter the state of the cluster. Since DDBMSs do not follow a unified standard for these commands, users must tailor the customization for each specific DDBMS. Based on our analysis, we found that migrating the tool to a MySQL-based distributed database (e.g., from MySQL NDB Cluster to MariaDB Galera Cluster) requires relatively

minor modifications, as these systems support MySQL-like SQL syntax. Specifically, commands that modify the cluster state need to be replaced, and relevant APIs must be adjusted, resulting in an estimated 20 lines of code changes. In contrast, migrating to non-MySQL distributed databases (e.g., from MySQL NDB Cluster to TiDB Cluster) necessitates additional adaptation for both the SQL sequence generation module and the cluster state change command module, leading to an estimated 500 lines of code changes. Finally, when migrating the tool to other distributed systems, further consideration of the synchronization logic is required—particularly regarding whether the target system employs a coordination mechanism similar to distributed databases, where management nodes and data nodes interact in a comparable manner. This characteristic determines whether our tool can partition the synchronization process for the target system in a similar way, thus identifying the temporal sequence of cluster state changes.

5 Evaluation

We examine DepState's capability to detect synchronization-failure bugs and produce complex data and high-quality state sequences. Our evaluation addresses these research questions:

- RQ1: Can DepState find new synchronization failure bugs in DDBMSs?
- RQ2: Can DepState perform better than other state-of-the-art testing techniques?
- RQ3: What are the contributions of each component in DepState?

5.1 Evaluation Setup

5.1.1 Tested DDBMSs. To evaluate the generality and efficiency of DepState, we select four popular open-source DDBMSs for evaluation, namely MySQL NDB Cluster [29], MySQL InnoDB Cluster [28], MariaDB Galera Cluster [5], and TiDB Cluster [31]. They are all widely used open-source DDBMSs according to DB-Engine Ranking [37]. They support similar DBMS features (e.g., the relational data model) but use different distributed architecture implementations. The versions under evaluation are MySQL NDB Cluster v8.0.40, MySQL InnoDB Cluster v8.0.35, MariaDB Galera Cluster v10.6.20, and TiDB Cluster 8.0.11-TiDB-v9.0.0-alpha-100-g91706ec8df-dirty.

5.1.2 Compared Techniques. To evaluate the effectiveness of DepState in testing DDBMSs, we compare it with five state-of-the-art tools: Jepsen [17], Mallory [26], SQLsmith [2], SQLancer [33], and Mozi [23]. Jepsen [17] is a distributed system testing tool, which has been adapted to test DDBMSs in the industry. Mallory [26] is an enhancement built upon the foundation of Jepsen. SQLsmith, SQLancer, and Mozi are three state-of-the-art DBMS testing techniques, which have detected hundreds of bugs in practice. We adapted Jepsen's default configuration to make it work on these DDBMSs, and Mallory was similarly configured. For SQLsmith, SQLancer, and Mozi, since they were designed for testing traditional DBMS, we used them on a single SQL server of the DDBMS as a baseline. We selected traditional database testing tools, including SQLsmith, SQLancer, and Mozi, as benchmarks. By using them as baselines, we aim to demonstrate that traditional database testing tools are not directly applicable to distributed databases.

5.1.3 Basic Setup. All experiments were conducted on a 64-bit Ubuntu 22.04 machine with an AMD EPYC 7742 processor (128 cores @ 2.25 GHz) and 488 GiB of main memory. Each testing tool used its default configuration to ensure consistency throughout all evaluations. The DDBMS clusters employed a single setup with 2 replicas and 2 slices, without multi-cluster failover. We ran each DDBMS with one testing tool for 24 hours, which is a commonly used time setup.

Table 2. Synchronization Failure Bugs Detected by DepState Within Two Weeks

#	DDBMSs	Bug Type	Root Cause	Bug Status
1	MySQL NDB Cluster	Crash	Engine mishandles metadata synchronization and locking.	Confirmed
2	MySQL NDB Cluster	Crash	Concurrent cluster restart during node reboot causes state inconsistency.	Confirmed
3	MySQL NDB Cluster	Crash	Improper lock handling during node removal in synchronization.	Confirmed
4	MySQL NDB Cluster	Crash	Internal error occurs during signal transmission in nodes.	Investigating
5	MySQL NDB Cluster	Crash	Improper metadata lock handling during synchronization with complex dependencies.	Confirmed
6	MySQL NDB Cluster	Crash	FindTable function failure during BACKUP.	Investigating
7	MySQL NDB Cluster	Crash	SimulatedBlock component's signal processing fails.	Confirmed
8	MySQL NDB Cluster	Crash	Data Check fails, the specified table or table pointer could not be found.	Investigating
9	MySQL NDB Cluster	Crash	SUMA bucket switch failure during asynchronous event processing.	Confirmed
10	MySQL NDB Cluster	Crash	Forced shutdown-induced signal processing error caused cascading node restarts.	Investigating
11	MySQL NDB Cluster	Crash	Some operations are not supported when synchronizing complex SQL queries.	Investigating
12	MySQL NDB Cluster	Hang	Timeout mechanism failure in NDB Cluster during complex query execution.	Confirmed
13	MySQL NDB Cluster	Hang	Failure in query plan generation and optimization when handling complex nested queries.	Confirmed
14	MySQL NDB Cluster	Hang	Transaction optimization enters an infinite loop.	Confirmed
15	MySQL NDB Cluster	Hang	ID allocation failure disrupts synchronization during node rejoin.	Confirmed
16	MySQL NDB Cluster	Hang	Synchronization fails during complex query processing.	Confirmed
17	MySQL NDB Cluster	Inconsistency	Failure to send synchronization signal in function.	Investigating
18	MySQL NDB Cluster	Inconsistency	Error occurred updating automatic index statistics.	Investigating
19	MySQL InnoDB Cluster	Crash	Incompatible data types cause synchronization to fail.	Investigating
20	MySQL InnoDB Cluster	Crash	A data type conversion error after a network connection failure causes the node to exit.	Investigating
21	MariaDB Galera Cluster	Crash	Missing records and duplicate key conflicts in delete and update operations.	Investigating
22	MariaDB Galera Cluster	Inconsistency	Data type mismatches due to invalid default values are ignored by WSREP.	Investigating
23	TiDB Cluster	Crash	Frequent PD-TiKV Connection Retries.	Investigating
24	TiDB Cluster	Crash	Negative expire-at Value Causes RPC Failures.	Investigating
25	TiDB Cluster	Crash	TSO Client Cancellation During Startup	Investigating

5.2 Synchronization Failure Bug Detection

We applied DepState to MySQL NDB Cluster[29], MySQL InnoDB Cluster[28], MariaDB Galera Cluster[5], and TiDB Cluster [31] to rigorously test synchronization failure bugs over a two-week period. These evaluated DDBMSs are widely adopted and have undergone extensive testing over decades. Nevertheless, DepState still demonstrated strong performance. Eleven synchronization failure bugs were confirmed as previously unidentified issues in these DDBMSs, and twelve are still under investigation due to the inherent challenges of reproducing bugs in distributed systems.

5.2.1 Overall Results. DepState has found a total of 25 previously unknown synchronization failure bugs on four well-tested DDBMSs within two weeks, with 13 confirmed by the developers. We also utilized additional tools to test these DDBMSs; however, none were able to detect any synchronization failure bugs. Table 2 shows DepState finds 18, 2, 2, and 3 bugs in MySQL NDB Cluster, MySQL InnoDB Cluster, MariaDB Galera Cluster, and TiDB Cluster, respectively. It indicates that DepState identifies bugs that lead to system crashes or hangs. The detected issues exhibit significant variety, encompassing a range of problems related to locking and synchronization, node management during reboots, query execution, aand optimization failures, as well as internal errors in signal handling. The results show that synchronization-oriented testing helps DepState to trigger synchronization failure bugs, which may lead to serious consequences. All identified bugs were reported to the corresponding DDBMS vendors, with 13 of these bugs being confirmed.

5.2.2 Impact of Found Synchronization Failure Bugs. We analyze the impact of synchronization failure bugs detected by DepState. Based on insights from DDBMS developers, 17 synchronization failure bugs identified by DepState are linked to distributed cluster states. Thirteen of these bugs cause complete SQL server crashes, resulting in service interruptions and compromising system availability. Four bugs cause data node crashes, leading to data loss, inconsistency, and potential node or cluster-wide outages. These synchronization failures critically undermine DDBMS stability.

5.2.3 Case Study. Crashes triggered by node removal during synchronization. DepState detected a synchronization failure bug which causes all data nodes in a MySQL NDB Cluster to crash. Figure 7

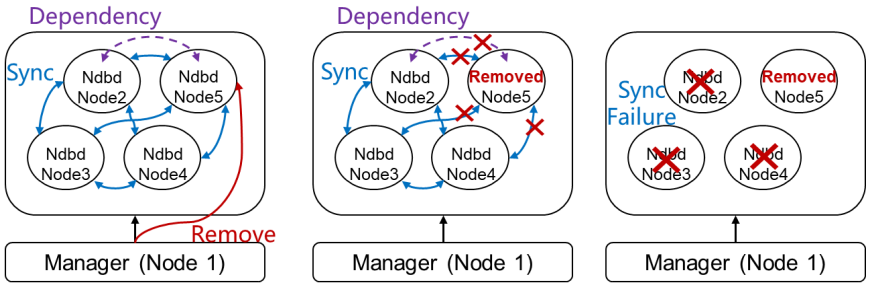


Fig. 7. MySQL NDB Cluster Crashes Triggered by Node Removal During Synchronization Process.

illustrates the occurrence of this bug within a cluster configuration consisting of one manager node (node1) and four ndbd data nodes (node2 through node5). The bug is triggered when node5 is removed by the cluster manager during the synchronization process on node2. Notably, node5 contains data dependencies related to node2. Once node5 is removed, these dependencies are severed, leading to a failure in the synchronization process on node2. This loss of dependencies causes the cluster to become unstable and ultimately results in a complete cluster crash.

In the report, DepState thoroughly documented the operations leading to the bug, including two main components: firstly, the specific workloads executed by each sql node, the precise start and end times of each SQL statement, and the corresponding outcomes of the workload executions; secondly, the management node's cluster operations and associated behavioral changes, along with their respective timings and results. This information effectively captured the particular scenario of the bug, outlining the sequence of operations and their timing for precise reproduction. Additionally, DepState preserved the database state at the time of the bug occurrence to facilitate accurate reproduction. Upon submission to MySQL's official bug report site, developers successfully reproduced and confirmed the bug. We present a detailed example in our repository.

Root cause of the bug. The issue stems from a locking error during synchronization. Specifically, node2 prematurely unlocks a critical synchronization lock on node5 due to a data dependency while processing node5's exit. This action does not wait for other nodes to finish their own synchronization, causing lock-state inconsistencies across the cluster. Consequently, these inconsistencies trigger the synchronization failure and lead to a cluster crash.

Why was the synchronization failure bug only found by DepState? Detecting this bug requires complex cross-node dependencies (from node5 to node2) triggered by cluster architecture changes during synchronization. DepState addresses this via ShardinOg-Aware Data Generation and Synchronization-Sensitive Operation Sequences, injecting fine-grained behaviors to reveal intricate synchronization issues. DepState successfully detected this vulnerability within hours, highlighting its efficacy in capturing complex synchronization behaviors. This demonstrates DepState's capability in discovering previously unknown *synchronization failure bugs*, answering **RQ1**.

5.3 Comparison with Other Techniques

To evaluate the effectiveness of our DepState, we conducted comparison experiments between DepState with Jepsen, Mallory, SQLsmith, SQLancer, and Mozi. Jepsen and Mallory are state-of-the-art tools for testing distributed systems. SQLsmith, SQLancer and Mozi are three popular DBMS testing methods. These five tools support testing the tested DDBMSs, and we compare DepState to them using their default configurations. We ran the testing tools on each DDBMS for

Table 3. Number of Unique Bugs Detected by Jepsen, Mallory, SQLsmith, SQLancer, Mozi, and DepState

DDBMS	Jepsen	Mallory	SQLsmith	SQLancer	Mozi	DepState
MySQL NDB Cluster	0	1	0	1	7	7
MySQL InnoDB Cluster	1	0	0	1	6	2
MariaDB Galera Cluster	1	1	0	1	0	2
TiDB Cluster	0	0	0	0	0	3
Total (Sync Failure Bugs)	2 (0)	2 (0)	0 (0)	3 (0)	13 (0)	14 (14)

24 hours, recording the number of detected bugs and covered synchronization-related functions as metrics. These functions are implemented specifically for the synchronization process in DDBMSs.

5.3.1 Bug Statistics. DepState outperforms other state-of-the-art testing techniques in finding DDBMS bugs. Table 3 displays the number of bugs detected by each tool in 24 hours. Specifically, DepState detected a total of 7, 2, 2, and 3 *synchronization failure bugs* on MySQL NDB Cluster, MySQL InnoDB Cluster, MariaDB Galera Cluster, and TiDB Cluster, respectively. In comparison, Jepsen, Mallory, SQLancer, and Mozi reported 2, 2, 3, and 13 bugs, respectively, while SQLsmith found no bugs. We analyzed the root causes of the bugs detected by each tool. The 14 bugs found by DepState are all related to the synchronization process. In contrast, the bugs identified by other tools are transaction or logic bugs, with none involving synchronization functions in DDBMS.

5.3.2 Synchronization-Related Function Coverage. One of the main reasons that DepState discovered *synchronization failure bugs* is that DepState can cover more synchronization function logic and trigger more behaviors of DDBMSs compared to Jepsen, Mallory, SQLsmith, SQLancer, and Mozi. Table 4 shows the number of synchronization-related function lines covered by each technique in 24 hours. It demonstrates that DepState covers more synchronization-related function lines compared to Jepsen, Mallory, SQLsmith, SQLancer, and Mozi. Specifically, DepState covers 6.13%-66.51%, 5.82%-57.28%, 14.12%-83.30%, 36.81%-83.88%, and 43.24%-54.28% more synchronization-related function lines compared to Jepsen, Mallory, SQLsmith, SQLancer, and Mozi, respectively.

The increased coverage of synchronization-related function lines is attributed to DepState's sharding-aware dependent data generation and cluster state sequence exploration. Sharding-aware data generation aids DepState in producing dependent test data, while cluster state sequence exploration triggers more synchronization behaviors. In contrast, Jepsen covers fewer synchronization function lines because its simplified approach lacks fine-grained analysis, modeling, and creation of complex data scenarios. Its fault injection is limited to the system level, without addressing deeper synchronization behaviors. Moreover, SQLsmith, SQLancer, and Mozi only generate SQL queries, hindering their ability to navigate the vast search space of distributed environments and detect cluster-state-induced bugs. In contrast, by using a designed data and mutation strategy, DepState discovered 14 synchronization failure bugs in 24 hours, showing DepState can exercise more synchronization functionality and find additional bugs, thus answering **RQ2**.

Table 4. Lines of Synchronization-Related Functions Covered by Tested Tools in 24 Hours

DDBMS	Jepsen	Mallory	SQLsmith	SQLancer	Mozi	DepState
MySQL NDB Cluster	11,850	11,092	11,505	10,588	10,622	**16,388**
MySQL InnoDB Cluster	2,096	2,219	1,904	2,551	2,731	**3,490**
MariaDB Galera Cluster	1,241	1,257	1,147	1,185	1,191	**1,706**
TiDB Cluster	4,127	4,139	3,838	2,382	3,002	**4,380**

Table 5. Number of Synchronization Failure Bugs and Synchronization-Related Function Lines Covered by DepState^{Data-}, DepState^{Seq-}, and DepState in 24 Hours

	Number of Synchronization Failure Bugs			Line Coverage of Sync-Related Functions		
DDBMS	DepState^{Data-}	DepState^{Seq-}	DepState	DepState^{Data-}	DepState^{Seq-}	DepState
MySQL NDB Cluster	4	2	7	15,252	12,920	16,388
MySQL InnoDB Cluster	1	0	2	1,765	2,992	3,490
MariaDB Galera Cluster	0	1	2	1,647	1,532	1,706

5.4 Effectiveness of Each Component

To understand the contributions of the two components in DepState managing complex data and generating state-altering behavior sequences, we implemented two ablation models: DepState^{Data-} and DepState^{Seq-}. DepState^{Data-} disables the shard-aware dependent data generation and replaces it with randomized data generation, which generates data randomly. DepState^{Seq-} disables the synchronization-sensitive cluster state sequence exploration, which randomly changes the state behavior at arbitrary time points. Table 5 shows the number of detected *synchronization failure bugs* and covered synchronization-related function lines by DepState^{Data-} and DepState^{Seq-} on testing three DDBMSs for 24 hours. From the table, we can see that compared with DepState^{Data-} and DepState^{Seq-}, DepState detects 6 and 8 more *synchronization failure bugs*, and covers 3.58%-97.73% and 11.36%-26.84% more synchronization-related function lines, respectively.

The result is reasonable because we design shard-aware dependent data generation and synchronization sensitive cluster state sequence exploration to trigger more behavior of synchronization process in DDBMSs. Specifically, in the absence of dependent data generation, DepState^{Data-} encounters limitations in the complexity of synchronization processes that can be triggered, leaving certain extreme cases difficult to explore. However, failures tend to arise in more intricate and extreme synchronization processes. Similarly, without cluster state sequence exploration, the DepState^{Seq-} produces a significant number of redundant data processing operations, thereby activating only a limited range of DDBMS behaviors. Therefore, both components are crucial for effectively exploring states within the synchronization process and successfully detecting synchronization failure bugs, thereby adequately answering **RQ3**.

6 Discussion

6.1 Generality and Flexibility of DepState

DepState can be adapted to other DDBMS systems within five steps. The steps are as: 1) Modify the interface for SQL generation so the generated SQL can be injected into the DDBMS. 2) Adapt the handling of cluster state transitions, due to the absence of a recognized specification for such behaviors across DDBMSs. 3) Integrate the source code of the database to be tested. 4) Test the cluster using DepState. 5) Evaluate the test results based on the tool's output and cluster logs.

Among these steps, the first and second require manual intervention. Different DDBMSs possess diverse API interfaces, with each database implementing commands for cluster state transitions in distinct ways. Furthermore, there is no universally recognized specification set similar to standard SQL across different vendors, rendering the automation of these two steps challenging. The third and fourth steps, however, are automated processes; with a relatively complete codebase, DepState can streamline the testing process. In the fifth step, the user must analyze the failure log file and sequence process to identify the nature of the error. Due to the complexities of distributed environments, errors often arise in intricate ways. While our tool can detect and record errors, determining their root causes relies on the user's analysis and experience.

6.2 Testing Non-relational DDBMSs

In our implementation, DepState is tailored for relational DDBMSs, and is not yet applicable to other types of DDBMSs. It stems from the unique characteristics of relational DDBMS, particularly sharding mechanisms and the intricate relationships and dependencies inherent in their data storage and transactional operations. These complexities significantly heighten error probability during the synchronization process. Consequently, our focus is on addressing two primary challenges faced by synchronization mechanisms in relational DDBMS: the construction of complex data dependencies and the frequent changes in cluster state. Non-relational DDBMSs are widely used, and we plan to extend DepState to test them. These systems, featuring distributed storage and dynamic scaling, exhibit error patterns and synchronization issues significantly distinct from relational DDBMSs. Differences in sharding strategies and consistency models may affect availability and consistency. Moreover, flexible data models complicate dependencies, hindering error detection.

To address these challenges, we plan to introduce a modular design in DepState that allows for adaptation to various types of DDBMSs. We will develop synchronization detection mechanisms specifically tailored for non-relational DDBMSs, taking into account their unique operational patterns and data relationships. Through these enhancements, we aim to improve the adaptability and effectiveness of DepState across different database environments, thereby providing more comprehensive support for the testing needs of multiple databases.

6.3 Testing Other Distributed Systems

Distributed systems are diverse, designed for specific applications with unique characteristics and synchronization mechanisms. DepState is tailored for DDBMS, emphasizing mechanisms that ensure data consistency across nodes. Provided the synchronization process involves a coordinating node and inter-node communication, DepState perform fine-grained phase segmentation to identify injection points for cluster state changes. The modifications required adapting the SQL input generation component (about 750 lines) and cluster behavior fuzzing component (around 400 lines).

In addition to the common synchronization process, other DBMS may also face scenarios involving system changes, such as deploying new services in a microservices architecture or taking servers offline for maintenance. These challenges are similar to those faced by synchronization processes in DDBMSs discussed in our paper. Our future work also intends to utilize DepState to test these analogous scenarios in other distributed systems.

7 Related Work

7.1 Distributed System Testing

Distributed system testing ensures stable, reliable operation across multiple nodes and components. Due to complexity, testing methods include chaos testing, fault injection, and system fuzzing. Chaos testing introduces unpredictable failures into a live environment to verify the system's resilience and self-healing capabilities. For example, Chaos Monkey [4] randomly shuts down virtual machines or containers in production to assess resilience. Gremlin [14] introduces failures like service terminations and network partitioning to simulate real-world faults. Fault injection is a methodology involving the deliberate introduction of specific fault types (e.g., network delays, service crashes, or data corruption) to evaluate a system's response and recovery mechanisms under fault conditions. For instance, Jepsen [17, 18] verifies the fault tolerance and consistency of distributed systems by simulating conditions like network partitions and node failures. Mallory[26] tests system defenses and data security by simulating man-in-the-middle attacks. Additionally, Phoenix[25] and Monarch [24] incorporate fuzzing to detect bugs in distributed systems.

However, these frameworks have shortcomings in handling complex data types and fine-grained synchronization in DDBMSs. In contrast, DepState detects synchronization failure bugs by establishing dependencies between distributed tables across nodes and shards, facilitating deeper insight into data interrelationships. By generating data from these dependencies, DepState more accurately simulates real scenarios. Additionally, it detects faults by exploring cluster operation sequences that alter cluster states during key synchronization phases. This approach uncovers potential bugs and clarifies how states and interactions cause synchronization failures.

7.2 DBMS Testing

DBMSs, being inherently complex software systems, are consequently susceptible to various issues. Traditional DBMS testing, aimed at ensuring the security and correctness of the system, primarily focused on the generation of complex SQL queries and the evaluation of their effects on system performance and correctness. Squirrel [41] introduces mutation-based fuzzing to detect memory safety bugs, which combines SQL query generation with coverage feedback. LEGO [22] improves fuzzing by generating SQL sequences through combinatorial logic. It focuses on pairing SQL types to enhance branch coverage. Griffin [9] adopts a grammar-free approach. It uses metadata graphs to guide mutations, creating more valid SQL statements and finding errors missed by grammar-based fuzzers. Furthermore, SQLancer [33] employs metamorphic testing methodologies [32] to detect logical bugs within DBMSs, proposing three distinct test oracles for the automatic detection of various correctness issues. Mozi [23] proposes a configuration-based equivalent transformations framework to detect bugs. Thanos [10] is testing against the database's storage engine. As well as some more work on the configuration in the database [13].

However, existing DBMS testing tools are inadequate for DDBMSs due to their inability to effectively manage the complexities of distributed architectures, such as handling multiple nodes, shards, and intricate data dependencies. In contrast, DepState is specifically designed to detect problems in the synchronization process of DDBMSs. It first generates dependent data across different shards and then systematically alters cluster states. This approach helps identify potential issues related to complex interdependencies and state changes during the synchronization process.

8 Conclusion

This paper introduces DepState, a testing framework specifically designed for the synchronization processes in DDBMSs. The framework first generates databases with intricate constraints and dependencies by analyzing the data sharding state. Then, it captures cluster operation windows and explores cluster state change sequences to find synchronization failure bugs. We evaluated DepState on 4 widely-used DDBMSs, where it outperformed existing state-of-the-art DDBMS testing tools like Jepsen. Notably, DepState uncovered 25 new *synchronization failure bugs*, with 13 of them confirmed by vendors. The remaining issues are currently under investigation by developers. Our future work will enhance DepState's ability to handle more complex data types and extend DepState to test other non-relational DDBMSs.

Data Availability

For data transparency and reproducibility, the prototype of DepState is available at https://github.com/FangCundi/DepState.

Acknowledgments

This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan Province (No.2023RC1001, 2023RC3012), and NSFC No.U2441238 and No.62202474.

References

[1] Raoof Altaher. 2024. Transparency Levels in Distributed Database Management System DDBMS. *Preprints* (April 2024). doi:10.20944/preprints202404.1327.v1

[2] Anse1. 2015. *sqlsmith – A Random SQL Query Generator.* https://github.com/anse1/sqlsmith A random SQL query generator.

[3] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. 2019. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly Media, Sebastopol, CA, USA.

[4] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. 2015. Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 371–372. doi:10.1145/2785956.2790038

[5] MariaDB Community. 2024. *MariaDB Galera Cluster.* https://mariadb.com/kb/en/galera-cluster/

[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. doi:10.1145/2491245

[7] Stephen Elliot. 2014. DevOps and the Cost of Downtime: Fortune 1000 Best Practice Metrics Quantified. *International Data Corporation (IDC)* (2014).

[8] Paul Ezhilchelvan, Amjad Aldweesh, and Aad van Moorsel. 2018. Non-Blocking Two Phase Commit Using Blockchain. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems* (Munich, Germany) *(CryBlock'18)*. Association for Computing Machinery, New York, NY, USA, 36–41. doi:10.1145/3211933.3211940

[9] 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. doi:10.1145/3551349.3560431

[10] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. 2024. THANOS: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1–12.

[11] Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 133–160. doi:10.1145/1188913.1188915

[12] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317. doi:10.1145/289.291

[13] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. *Proc. VLDB Endow.* 16, 7 (March 2023), 1712–1725. doi:10.14778/3587136.3587145

[14] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 57–66. doi:10.1109/ICDCS.2016.11

[15] Shawn Hogan, Frazer Clement, and Jon Stephens. 2020. *Bug #98526 Forced Shutdown at End of Process When Data Node Tries to Join Cluster.* https://bugs.mysql.com/bug.php?id=98526

[16] Luke Jones. 2020. *Google Explains Monday's Google Cloud Outage That Took Down Gmail and YouTube.* https://winbuzzer.com/2020/12/15/google-explains-mondays-google-cloud-outage-that-took-down-gmail-and-youtube-xcxwbn/

[17] Kyle Kingsbury. 2023. *Jepsen.* https://jepsen.io/

[18] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *arXiv preprint* (2020). doi:10.48550/arXiv.2003.10554 arXiv:arXiv:2003.10554

[19] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: an overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware* (Yorktown Heights, New York, USA) *(LADIS '08)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. doi:10.1145/1529974.1529979

[20] Leslie Lamport. 2019. *The part-time parliament.* Association for Computing Machinery, New York, NY, USA, 277–317. https://doi.org/10.1145/3335772.3335939

[21] Sami M. Lasassmeh and James M. Conrad. 2010. Time synchronization in wireless sensor networks: A survey. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*. 242–245. doi:10.1109/SECON.2010.5453878

[22] 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)*. 668–681. doi:10.1109/ICDE55515.2023.00057

[23] 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 (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 135, 12 pages. doi:10.1145/3597503.3639112

[24] 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)*. USENIX Association, Santa Clara, CA, 529–543. https://www.usenix.org/conference/atc24/presentation/lyu

[25] Fuchen Ma, Yuanliang Chen, Yuanhang Zhou, Jingxuan Sun, Zhuo Su, Yu Jiang, Jiaguang Sun, and Huizhong Li. 2023. Phoenix: Detect and Locate Resilience Issues in Blockchain via Context-Sensitive Chaos. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1182–1196. doi:10.1145/3576915.3623071

[26] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1615–1629. doi:10.1145/3576915.3623097

[27] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[28] Oracle. 2024. *MySQL InnoDB Cluster Documentation*. https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html

[29] Oracle. 2024. *MySQL NDB Cluster Documentation*. https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html

[30] M. Tamer Özsu and Patrick Valduriez. 1999. *Principles of Distributed Database Systems*. Springer, Berlin, Heidelberg, Germany.

[31] PingCAP. 2024. *TiDB: An Open Source Distributed SQL Database*. https://pingcap.com/

[32] 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. doi:10.1145/3368089.3409710

[33] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 38, 16 pages. doi:10.5555/3488766.3488804

[34] Mark Roantree. 2002. Metadata Management in Federated Multimedia Systems. In *Australasian Database Conference*. 147–155.

[35] Amanpreet Kaur Sandhu. 2022. Big data with cloud computing: Discussions and challenges. *Big Data Mining and Analytics* 5, 1 (2022), 32–40. doi:10.26599/BDMA.2021.9020016

[36] Fred B. Schneider. 1982. Synchronization in Distributed Programs. *ACM Trans. Program. Lang. Syst.* 4, 2 (April 1982), 125–148. doi:10.1145/357162.357163

[37] Redgate Software. 2024. *DB-Engines Ranking*. https://db-engines.com/en/ranking

[38] Siamak Solat. 2024. Sharding Distributed Databases: A Critical Review. *arXiv preprint* (2024). doi:10.48550/arXiv.2404.04384 arXiv:arXiv:2404.04384

[39] Jason Warner. 2018. *October 21 Post-Incident Analysis*. https://github.blog/news-insights/company-news/oct21-post-incident-analysis/

[40] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. 2000. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. 464–474. doi:10.1109/ICDCS.2000.840959

[41] 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. doi:10.1145/3372297.3417260

Received 2024-10-31; accepted 2025-03-31