# State-Induced Approximate Metamorphic Testing for Detecting Logical Bugs in DBMSs

Li Lin*, Yaorui Fei*,, Lingfeng Bao*, Rongxin Wu†, Dongxiang Zhang*
*Zhejiang University, Hangzhou, China
{linli1210, fyr031109, lingfengbao, zhangdongxiang}@zju.edu.cn
†Xiamen University, China
wurongxin@xmu.edu.cn

***Abstract*—DBMS correctness depends not only on individual SQL queries, but also on the proper maintenance of database states, including schemas, data instances, and metadata. Incorrect handling of such state can cause state-related logical bugs, which silently produce wrong query results or inconsistent state changes. Existing DBMS testing techniques mainly transform queries over fixed states or check equivalence between transformed states, limiting their ability to expose bugs caused by schema-data interactions, constraint handling, data distributions, and optimizer behavior. In this paper, we propose *State-Induced Approximate Metamorphic Testing*, a new paradigm that induces approximate metamorphic relations through structured database-state transformations. We implement SCHEMAMORPH, which constructs related states via tuple-based and admissibility-based mutations, executes relation-preserving DQL/DML observations, and checks whether the observed outcomes preserve the induced approximation relations. Evaluated on five widely used DBMSs, SCHEMAMORPH reports 21 previously unknown state-related logical bugs, all of which have been confirmed.**

## I. INTRODUCTION

Database Management Systems (DBMSs) are critical infrastructure for modern data-driven applications, ranging from e-commerce and online banking to large-scale enterprise systems [1], [2]. They provide standardized interfaces such as SQL for defining schemas, modifying data, and retrieving query results [3]. A database state consists of the schema, the concrete data instance, and relevant metadata maintained by the DBMS [4], [5]. Together, these components determine how data is stored, validated, optimized, queried, and modified. Therefore, DBMS correctness depends not only on correctly interpreting individual SQL queries, but also on correctly maintaining valid database states and their semantic evolution under schema constraints and data modifications.

Due to the complexity of database schemas, data instances, and the intricate optimization mechanisms used by DBMSs [6], [7], it is challenging for relational DBMSs to correctly maintain and reason about the semantic relationships between schema-level information and concrete data states. Errors in handling such relationships, including constraints, nullability, data types, and value distributions, can introduce logical bugs into the system. These bugs can manifest in various forms, including incorrectly enforced constraints, wrong query results under specific data distributions, and inconsistent database states after data modification [8]–[10]. Such logical bugs are particularly dangerous because they do not trigger crashes or explicit failures, but instead silently produce incorrect query outputs or inconsistent states, often remaining undetected for long periods [11]–[13].

Furthermore, automatically exposing such bugs is highly challenging because they often emerge only under specific database states and DBMS interactions. For example, certain SQL queries may trigger wrong results only when their predicates, joins, type conversions, or execution plans interact with particular data distributions, indexes, constraints, or metadata. Therefore, detecting state-related logical bugs requires both testing strategies that systematically exercise such state interactions and test oracles that determine whether the observed behaviors are semantically valid. In this work, we focus on this class of bugs, which we refer to as *state-related logical bugs* (*StateLogicalBug*s), where incorrect handling of database-state information leads to erroneous query results or inconsistent state changes.

Figure 1 shows a real-world *StateLogicalBug* in MySQL (Bug #120253 [14]), where the same JOIN query produces an incorrect empty result after a state update. In the original state, invalid date inputs in `t_main` are stored as the zero date `'0000-00-00'`, and non-date `LONGTEXT` values in `t_ref` are implicitly converted to the same zero date when evaluating the predicate `m.d = r.lt`. As a result, the original query produces non-empty JOIN results. After inserting additional valid date tuples and refreshing table statistics, however, the same query unexpectedly returns zero rows, even though the previously matching data still exist. This is a logical bug because MySQL silently changes the query result due to an optimizer plan switch: the nested-loop index-lookup path fails to handle the implicit `LONGTEXT`-to-`DATE` conversion consistently with the hash-join path.

To detect logical bugs in DBMSs, existing approaches typically generate SQL statements, execute them on DBMSs, and check whether the produced results satisfy certain correctness expectations [8], [11], [13], [15]–[21]. A fundamental challenge in this process is the test oracle problem: for an automatically generated query, it is usually difficult to know the correct result in advance [22]. Existing approaches mainly address this challenge through two paradigms: differential

Lingfeng Bao is the corresponding author and is also affiliated with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

```
CREATE TABLE t_main (id INT PRIMARY KEY, d  DATE NOT NULL);
CREATE INDEX idx_d ON t_main (d);
CREATE TABLE t_ref (id INT PRIMARY KEY, lt LONGTEXT);

-- Original State S1(t_main 1 row,    t_ref 100 rows)
INSERT IGNORE INTO t_main VALUES (1, 'not-a-date');
INSERT INTO t_ref WITH RECURSIVE seq(n) AS (
   SELECT 1 UNION ALL
   SELECT n + 1 FROM seq WHERE n < 100
)  SELECT n, CONCAT('sample_', n) FROM seq;
ANALYZE TABLE t_main;    ANALYZE TABLE t_ref;

SELECT COUNT(*) AS cnt FROM t_main m JOIN t_ref  r ON m.d = r.lt;
--result:     100

-- Mutated State S2(t_main 2001 rows,    t_ref 100 rows)
SET SESSION cte_max_recursion_depth = 10000;
INSERT INTO t_main WITH RECURSIVE seq(n) AS (
   SELECT 4  UNION ALL
   SELECT n + 1 FROM seq WHERE n < 2003
) SELECT n, DATE_ADD('2000-01-01', INTERVAL n DAY) FROM seq;
ANALYZE TABLE t_main;

SELECT COUNT(*) AS cnt FROM t_main m JOIN t_ref  r ON m.d = r.lt;
--result:     >=100   --(expected)
--result:     0   --(actual)
```

Fig. 1: **A state-related logical bug in MySQL detected by SchemaMorph. (Bug #120253 [14])**

testing and metamorphic testing. Differential testing executes the same queries across multiple DBMSs, and treats inconsistent outputs as potential bugs [23], [24]. However, since dialects and state-level semantics are difficult to align across DBMSs, differential testing often has to restrict test cases to a common SQL subset, thereby weakening its ability to trigger *StateLogicalBug*s [19], [21].

To overcome these limitations, metamorphic testing (MT) has been widely adopted for DBMS logical bug detection. Instead of relying on cross-DBMS consistency, MT constructs mutated test cases within the same DBMS from an original test case and checks whether their outputs satisfy a predefined metamorphic relation (MR) [8], [11], [13], [15]–[21]. Existing DBMS-oriented MT techniques mainly fall into two categories. The first category focuses on query-level MRs, where the database state is fixed and only the SQL query is transformed [8], [13], [16]–[21]. Representative approaches, such as NoREC [11] and TLP [15], construct semantically related queries and check equivalence or approximate relations. Although effective for detecting many optimizer- and predicate-related bugs, these methods mainly exercise query semantics while leaving the underlying database state unchanged. They therefore have limited ability to expose *StateLogicalBug*s caused by interactions among schemas, constraints, data distributions, and state evolution. The second category studies state-level MRs, which transform state-level information such as schema definitions, Data Definition Language (DDL) sequences, or DBMS-maintained metadata [8]–[10]. For example, DDLCheck [9] constructs equivalent databases using different DDL sequences, while Radar [8] builds a raw database by removing metadata such as constraints or indexes from a metadata-rich database while preserving the same data. Although these approaches explicitly exercise schema- or metadata-related behaviors, their oracles are still based on state equivalence: the transformed database is expected to preserve the same observable semantics as the original one. Under this equivalence requirement, the state transformation space is limited, and the original and transformed states may still follow the same faulty state-reasoning path, producing identical but incorrect results.

Inspired by Pinolo [19] and ValScope [21], which demonstrate that approximate semantic relations can serve as effective oracles for logical bug detection, we revisit this principle from the perspective of database states. In this paper, we propose ***S**tate-**I**nduced **A**pproximate **M**etamorphic **T**esting* (**SIAMT**) for detecting *StateLogicalBug*s. Our key insight is that approximate metamorphic relations need not be induced only by transforming DQL queries; they can also be induced by structured transformations over database states. Specifically, SIAMT mutates database states by changing state-level information such as schema definitions, constraints, metadata, or data instances. These mutations are designed to create directional state relations rather than equivalent states. For example, a mutated state may contain fewer admissible tuples, less precise information, or a larger space of valid records, naturally inducing over- or under-approximation relations. SIAMT then applies the same DQL or DML observation to both the original and mutated states, and checks whether the two observable outcomes satisfy the expected approximation relation. If the observed query results or state changes violate this relation, the DBMS likely mishandles database-state information, thereby exposing *StateLogicalBug*s. For example, in Figure 1, SIAMT mutates the original database state by appending additional valid date tuples to `t_main`, while preserving all existing tuples that contribute to the original JOIN result. This state transformation induces an over-approximation relation: the mutated state contains all relevant data from the original state, and thus the result cardinality of the same JOIN observation should not decrease. SIAMT then executes the same JOIN query on both states and compares their observable results under this induced relation. Since the mutated state returns zero rows while the original state returns non-empty results, SIAMT detects a violation of the expected approximation relation and reports it as a potential *StateLogicalBug*.

To instantiate SIAMT, we develop SchemaMorph, a state-induced approximate MT framework for detecting *StateLogicalBug*s. SchemaMorph first constructs an initial database state with schemas, data instances and then derives a related mutated state through two complementary strategies. Tuple-based mutation extends the original state by append-only data insertion, while admissibility-based mutation relaxes schema constraints and replays the same workload to admit additional tuples. After constructing such a pair of related states, SchemaMorph executes deterministic DQL and DML observations on them to examine whether the induced approximation relation is preserved. It then extracts comparable outcomes from query results or DML-induced state effects, and checks whether the outcomes over the mu-

tated state preserve those over the original state. Additionally, SCHEMAMORPH also uses distribution shaping, statistics refresh, plan feedback, and relaxation-aware observation biasing to increase the likelihood of exposing *StateLogicalBug*s.

We evaluate SCHEMAMORPH on five widely used and extensively tested DBMSs: MySQL, MariaDB, Percona, PolarDB, and OceanBase. Overall, SCHEMAMORPH reported 21 previously unknown *StateLogicalBug*s, including 8 in MySQL, 2 in MariaDB, 7 in Percona, 3 in PolarDB, and 1 in OceanBase; all of them have been confirmed by developers. Comparative studies show that SCHEMAMORPH detects bugs missed by representative DBMS testing tools, while ablation studies confirm that its optimization strategies improve both query plan diversity and bug-triggering capability. Overall, we make the following contributions:

- We propose *State-Induced Approximate Metamorphic Testing*, a new testing paradigm for detecting state-related logical bugs in DBMSs.
- We implement SCHEMAMORPH, a practical framework that constructs initial database states, derives related mutated states, executes DQL/DML observations, and checks whether the observed outcomes preserve the induced approximation relations.
- We evaluate SCHEMAMORPH on five widely used DBMSs. In total, SCHEMAMORPH reports 21 previously unknown *StateLogicalBug*s, all of them have been confirmed.

## II. BACKGROUND AND MOTIVATION

### A. Background

**Relational Database Management Systems.** Relational Database Management Systems (DBMSs) are used to create, manage, and query relational databases [25]. In a relational DBMS, data is organized into structured tables (relations) with predefined columns and rows, enabling efficient storage, retrieval, and manipulation. Structured Query Language (SQL) is the standard language for interacting with relational DBMSs, supporting data definition, manipulation, and querying [26]. In this work, we focus on relational DBMSs. Unless otherwise specified, we use the term "DBMS" to refer to relational DBMSs throughout the paper.

**Structured Query Language.** Relational DBMSs provide SQL as a unified interface for interacting with stored data, enabling a wide range of operations such as schema definition, data modification, and data retrieval. Based on their functionality, SQL statements are categorized into three main types.

*Data Definition Language* (DDL) statements are responsible for defining and modifying database schema objects. Typical examples include `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE`, which are used to create new relations, adjust existing structures, and remove schema objects, respectively.

*Data Manipulation Language* (DML) statements are designed for modifying the data stored in database tables. Representative operations include `INSERT` for adding new tuples, `UPDATE` for modifying existing records, and `DELETE` for removing tuples from a table.

*Data Query Language* (DQL) statements are used for retrieving data from the database. The most representative DQL statement is `SELECT`, which supports a rich set of query constructs, including filtering conditions (e.g., `WHERE`), ordering (`ORDER BY`), aggregation functions, join operations, and grouping (`GROUP BY`), enabling flexible and expressive data analysis over one or multiple tables.

**Database State.** A database state refers to a snapshot of a database at a given time [27]. It includes the database schema, the concrete data stored in the tables, and relevant metadata maintained by the DBMS. The schema defines the logical structure of the database, such as tables, columns, data types, and integrity constraints, while the data instance contains the actual tuples that follow these definitions. Metadata records system-maintained information about schemas and data, such as catalog entries, indexes, constraints, statistics, and storage-related information, which may affect validation, optimization, and query execution. As SQL statements are executed, the database may evolve from one state to another. For example, Data Definition Language (DDL) statements can change the schema or metadata, Data Manipulation Language (DML) statements can change the stored data, and these changes may affect the behavior of later queries or updates. Therefore, manipulating database states provides a natural way to exercise the interactions among schemas, metadata, constraints, data values, and query execution.

**Query Plan.** A query plan consists of a sequence of logical and physical operators, such as table scans, index lookups, joins, and aggregations, along with their execution order [28], [29]. The DBMS optimizer is responsible for selecting an efficient query plan based on available schema information and data statistics. The choice of query plan is highly dependent on the database state. In particular, schema properties (e.g., indexes and constraints) and data characteristics (e.g., cardinality and value distribution) can significantly affect optimization decisions [30], [31]. As a result, even when executing the same query, different database states may lead to different query plans, which in turn may expose different execution behaviors. To reason about query plans, DBMSs typically provide diagnostic interfaces such as `EXPLAIN`, which outputs the execution plan chosen by the optimizer without actually executing the query. In addition, many DBMSs maintain internal statistics to support query optimization. For example, the `ANALYZE TABLE` command updates statistics such as table cardinalities and value distributions, which are used by the optimizer to estimate costs and select execution plans.

**State-related Logical Bugs.** State-related logical bugs arise from the complexity of managing database states, where errors occur when the DBMS fails to maintain correct semantic relationships between schema, data, and constraints. These logical bugs can manifest in two major forms: DQL queries may return incorrect results over certain database states, and DML operations may produce incorrect or inconsistent database states after execution [8]–[10]. Their impact is significant, as they can silently corrupt the database, affecting future queries or operations without triggering system failures. Detecting

TABLE I: **Comparison of representative DBMS-oriented metamorphic testing approaches.**

| Approaches / Tools | Venues | Transformation Target | MR Semantics |
|---|---|---|---|
| NoREC [11] | ESEC/FSE'20 | SELECT statement form | Equivalence |
| TLP [15] | OOPSLA'20 | SELECT predicates / partitioned SELECTs | Equivalence |
| Pinolo [19] | ATC'23 | SELECT predicates | Approximation |
| EET [18] | OSDI'24 | SELECT query expressions | Equivalence |
| CODDTest [32] | SIGMOD'25 | SELECT query expressions | Equivalence |
| ValScope [21] | OSDI'26 | SELECT query expressions / value computation | Approximation |
| Radar [8] | PVLDB'24 | Metadata | Equivalence |
| DDLCheck [9] | PVLDB'25 | DDL sequences | Equivalence |
| **This work** | – | **Database states** | **Approximation** |

**Note.** For SELECT-level approaches, the transformation target refers to components or forms of SELECT statements, such as predicates and expressions, rather than the entire SELECT statement.

these bugs is challenging because they do not produce explicit errors and often require inspecting the evolution of the database state across transformations. In this work, we refer to this class of bugs as *state-related logical bugs* (*StateLogicalBug*s), where incorrect handling of database-state information leads to erroneous query results or inconsistent state changes.

### *B. Limitations of Existing Metamorphic Testing Approaches*

Metamorphic testing (MT) detects logical bugs by generating mutated queries from original queries and checking predefined metamorphic relations (MRs) over their outputs. Although existing MT approaches have been effective in exposing logical bugs, their testing capability largely depends on two design choices: what is transformed and what relation is checked. Table I summarizes representative DBMS-oriented MT approaches from these two dimensions.

As shown in Table I, most existing MT approaches transform SELECT-level constructs, such as statement forms, predicates, or query expressions. These approaches define equivalence or approximation relations over query results, making them effective for testing SELECT query processing, predicate evaluation, expression computation, and optimizer behavior. However, they largely keep the underlying database state unchanged, and thus have limited ability to expose bugs that require changing schema information, metadata, constraints, data distributions, or state evolution. Recent state-level approaches, such as DDLCheck [9] and Radar [8], transform DDL sequences or metadata, but their MRs are still based on equivalence: the transformed state is expected to preserve the same observable semantics as the original state. This equivalence requirement restricts the state transformation space, since the original and transformed states may still follow the same faulty state-reasoning path and produce identical but incorrect results. Therefore, existing MT approaches leave an unexplored design space: approximation relations induced by database-state transformations, which are essential for exposing *StateLogicalBug*s caused by incorrect reasoning about schema–data interactions and state evolution.

### *C. Key Insights*

The limitations above suggest a missing point in the design space of DBMS-oriented MT: existing approaches either induce approximation relations by transforming DQL queries, or transform database states only under equivalence-based oracles. Our key insight is that approximate MRs can also be induced by structured transformations over database states. Based on this insight, we propose ***S**tate-**I**nduced **A**pproximate **M**etamorphic **T**esting* (**SIAMT**) for detecting *StateLogicalBug*s. Instead of treating the database state as a fixed input, SIAMT constructs related database states by changing schema definitions, constraints, metadata, or data instances. These states are designed to satisfy directional approximation relations, such as over- or under-approximation, rather than strict equivalence. However, state mutations alone are not sufficient to expose *StateLogicalBug*s, because such bugs often arise when the DBMS interprets, optimizes, queries, or updates the related states. Therefore, SIAMT applies the same DQL query or DML operation to both the original and mutated states, and checks whether their observable outcomes preserve the approximation relation induced by the state transformation. A violation indicates that the DBMS may incorrectly maintain, interpret, or optimize database-state information. The central challenge is to jointly design state mutations that induce predictable approximation relations and observations that can effectively expose violations of these relations.

### *D. Challenges*

**Challenge #1: Constructing effective approximate states.** Applying SIAMT is challenging because not all database-state transformations induce valid and useful approximation relations. For example, arbitrary changes to tuples, constraints, or metadata may break schema compatibility, make observations incomparable, or produce state pairs whose semantic relation is unclear. Meanwhile, the space of possible state transformations is extremely large: even simple choices, such as which table to modify, which columns or constraints to change, and how many tuples to add, remove, or reject, can lead to many possible state pairs and approximation directions. Blindly exploring this space is inefficient and may produce state pairs that are semantically weak, difficult to observe, or unlikely to exercise interesting DBMS reasoning paths. Thus, SIAMT must efficiently construct state pairs that induce clear approximation relations and are effective for bug detection.

**Challenge #2: Soundly observing approximation relations.** Even when two database states satisfy an intended approximation relation, arbitrary DQL/DML statements may not be suitable for observing it. Some statements may contain nondeterministic behavior, causing unstable outcomes even over the same state. Others may involve non-monotonic constructs or global dependencies, such as ordering, ranking, or result truncation, which can break subset or monotonicity relations even when the DBMS is correct. Thus, SIAMT must carefully select observations whose results or state changes are deterministic and relation-preserving across related states. For example, Figure 2 shows a case where the mutated state $S_2$ extends the original state $S_1$ by adding one tuple $(4, 200)$, so the two states satisfy $S_1 \subseteq S_2$. However, the query with "`ORDER BY price DESC LIMIT 2`" does not preserve this subset relation, because the new high-price tuple evicts an

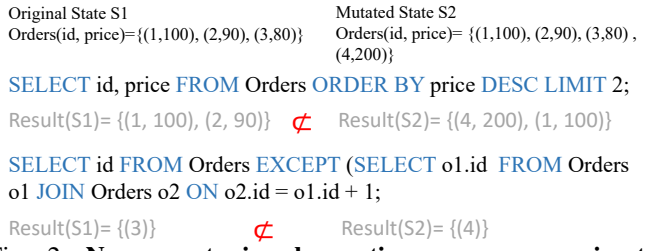


Fig. 2: **Non-monotonic observations over approximate database states.**

existing tuple from the limited result. Similarly, the `EXCEPT` query returns tuples whose `id` values have no successor in the same table; after adding the tuple $(4, 200)$, the previously returned tuple with $id = 3$ disappears from the result because it now has a successor. Thus, even when database states satisfy a clear approximation relation, observation statements with global ordering, result truncation, or anti-monotonic predicates may fail to preserve the expected relation.

### E. Solutions

**Solution #1: Semantics-guided and plan-aware state transformation.** To construct effective approximate states, SIAMT avoids arbitrary database-state mutations and instead organizes transformations around two approximation principles: tuple-induced approximation and admissibility-induced approximation. The former changes the admissible tuple set while preserving schema compatibility, and the latter changes schema properties, constraints, or value admissibility while keeping the same observations executable across related states. These transformations induce clear over- or under-approximation relations, which provide the semantic basis for reliable metamorphic oracles. To make these relations more effective for bug detection, SIAMT amplifies state differences along the transformed dimension, such as data distributions, boundary values, constraint admissibility, and index-related properties. Moreover, SIAMT uses lightweight query plan feedback to prioritize state pairs that are more likely to exercise different optimization or execution paths. Together, these strategies allow SIAMT to efficiently explore the large state-transformation space while preserving meaningful approximation relations; Section IV details the concrete transformation strategies and optimizations.

**Solution #2: Relation-aware and monotonic observation.** To soundly observe approximation relations, SIAMT does not apply arbitrary DQL or DML statements to related states. Instead, it selects observations according to the semantic direction induced by each state transformation, ensuring that the observation itself preserves the expected relation. For DQL observations, SIAMT focuses on deterministic and monotonic query patterns, and avoids constructs that may break relation preservation, such as nondeterministic expressions, result truncation, global ordering dependencies, and anti-monotonic predicates. For DML observations, SIAMT restricts operations to those whose state effects remain comparable across related states, so that successful executions, affected tuples, or resulting state changes follow the induced approximation relation. Together, these relation-aware restrictions allow SIAMT to distinguish genuine violations of database-state reasoning from artifacts introduced by unsuitable observations.

### F. Motivating Example

Figure 1 shows a real MySQL bug detected by our approach. In the original state $S_1$, SIAMT creates two tables, where `t_main.d` is an indexed `DATE` column and `t_ref.lt` is a `LONGTEXT` column; after inserting one invalid date into `t_main` using `INSERT IGNORE` and 100 non-date strings into `t_ref`, the join query returns 100 rows. SIAMT then constructs a mutated state $S_2$ by appending 2,000 valid-date rows to `t_main` and refreshing optimizer statistics, so $S_2$ is a tuple-level over-approximation of $S_1$. The observation query is a deterministic positive inner-join count query, without result truncation, anti-monotonic predicates, or nondeterministic expressions; therefore, it should preserve the tuple-level approximation relation. Since the transformation is append-only and `t_ref` remains unchanged, the result count in $S_2$ should be no smaller than that in $S_1$. However, MySQL returns 0 rows in $S_2$, violating the expected monotonicity relation. Further inspection shows that the state expansion changes the execution plan from a hash join to an index-lookup plan, where MySQL mishandles the required `LONGTEXT`-to-`DATE` coercion; forcing the old plan recovers the missing rows, confirming that the violation exposes a *StateLogicalBug*.

## III. State-Induced Approximate Metamorphic Testing

We now present ***S**tate-**I**nduced **A**pproximate **M**etamorphic **T**esting* (**SIAMT**), the core testing principle underlying SchemaMorph. The key idea of SIAMT is to induce directional approximation relations by transforming database states. Given two related states, SIAMT applies the same observation to both states and checks whether their observable outcomes preserve the approximation relation induced by the state transformation. If the observed outcomes violate the expected relation, the DBMS may introduce a *StateLogicalBug*. In the following, we formalize these concepts and then explain how they lead to the SIAMT oracle.

### A. Database States and Observable Outcomes

Let $S$ denote a database state. A database state consists of a schema, a concrete data instance, and DBMS-maintained metadata:

$$S = (\Sigma, I, M),$$

where $\Sigma$ denotes schema-level information, such as tables, columns, data types, constraints, and indexes; $I$ denotes the concrete tuples stored in the database; and $M$ denotes metadata maintained by the DBMS, such as catalog entries, statistics, and optimizer-related information.

In SIAMT, DQL and DML statements serve as observations over database states. A DQL observation exposes query results, while a DML observation exposes state effects caused by data modification. To uniformly compare these different

forms of observations, we define an abstract outcome function that captures what the oracle compares.

**Definition III.1** (Observable Outcome)**.** Given an observation $o$ and a database state $S$, the observable outcome of executing $o$ over $S$ is denoted by

$$\Omega(o, S).$$

For a DQL observation, $\Omega(o, S)$ represents an abstract, comparable representation of the query result, such as a result multiset, a row-token set, a cardinality, or an aggregate summary. For a DML observation, $\Omega(o, S)$ represents an abstract state effect induced by executing $o$, such as the set of rows inserted, updated, or deleted.

SIAMT does not require the exact correct output of an observation to be known in advance. Instead, it only requires that the outcomes of the same observation over two related states can be compared under an expected relation.

### *B. State-Induced Approximation Relations*

A state transformation may establish a directional semantic relation between two database states. For example, a transformed state may contain all tuples of the original state and additional tuples, or it may admit all tuples accepted by the original state under a shared workload and additional tuples rejected by the original state. In such cases, the transformed state becomes an over-approximation of the original state along the transformed state dimension.

**Definition III.2** (State-Induced Approximation)**.** Given a state transformation $\tau$ and two database states $S_1$ and $S_2 = \tau(S_1)$, we say that $\tau$ induces an approximation relation from $S_1$ to $S_2$, denoted by

$$S_1 \sqsubseteq_\tau S_2,$$

if $S_2$ preserves the information in $S_1$ that is relevant to a supported class of observations, while possibly admitting additional tuples, values, or state effects along the transformed state dimension. In this case, $S_1$ is an under-approximation of $S_2$, and $S_2$ is an over-approximation of $S_1$.

The relation $\sqsubseteq_\tau$ is parameterized by the transformation $\tau$ because the meaning of approximation depends on how the transformed state is constructed. Different transformations induce approximation over different semantic objects. For example, an append-only transformation induces containment over table tuples, while a constraint-relaxing transformation induces containment over the tuples admitted by a shared workload. Based on these two forms of containment, SIAMT considers two representative forms of state-induced approximation: tuple-induced approximation and admissibility-induced approximation.

**Tuple-induced approximation.** The first form directly changes the tuple set while preserving schema compatibility.

**Definition III.3** (Tuple-Induced Approximation)**.** Let $S_1 = (\Sigma, I_1, M_1)$ and $S_2 = (\Sigma, I_2, M_2)$ be two database states with the same schema $\Sigma$. We say that $S_1$ is a tuple-induced under-approximation of $S_2$, denoted by

$$S_1 \sqsubseteq_{\mathsf{tuple}} S_2,$$

if for every transformed table $T$,

$$I_1(T) \subseteq I_2(T),$$

and for every untransformed table $T'$,

$$I_1(T') = I_2(T').$$

Intuitively, $S_2$ extends $S_1$ by adding tuples without removing or modifying the tuples already present in $S_1$. Therefore, for observations whose semantics are monotonic with respect to tuple extension, the observable outcome over $S_2$ should not lose the outcome already observed over $S_1$. For example, if a deterministic positive selection or join returns a set of row tokens over $S_1$, executing the same observation over $S_2$ should return a superset of those row tokens.

**Admissibility-induced approximation.** The second form changes the admissibility conditions of database states while preserving the same relational interface. Two states may expose the same tables and columns, but differ in constraints such as nullability, uniqueness, check constraints, or foreign keys. Such differences induce an approximation relation over the tuples accepted by the two states after replaying the same data-modification workload.

**Definition III.4** (Admissibility-Induced Approximation)**.** Let $S_1 = (\Sigma_1, I_1, M_1)$ and $S_2 = (\Sigma_2, I_2, M_2)$ be two database states that expose the same relational interface, and let $W$ be a shared data-modification workload replayed over both states. We say that $S_1$ is an admissibility-induced under-approximation of $S_2$, denoted by

$$S_1 \sqsubseteq_{\mathsf{adm}} S_2,$$

if every tuple admitted by $S_1$ under $W$ is also admitted by $S_2$:

$$\mathsf{Adm}(S_1, W) \subseteq \mathsf{Adm}(S_2, W).$$

Here, $\mathsf{Adm}(S, W)$ denotes the set of row tokens admitted into state $S$ after replaying workload $W$. For example, if $S_1$ enforces a `NOT NULL`, `UNIQUE`, `CHECK`, or foreign-key constraint that is relaxed in $S_2$, the same insert workload may cause $S_1$ to reject some tuples that are accepted by $S_2$. In this case, the accepted tuple set of $S_1$ is contained in that of $S_2$, so $S_1 \sqsubseteq_{\mathsf{adm}} S_2$.

### *C. Relation-Preserving Observations*

A state-induced approximation relation alone is not sufficient to form a sound oracle. Even when $S_1 \sqsubseteq_\tau S_2$ holds, an arbitrary observation may not preserve this relation. For example, observations involving nondeterministic expressions, result truncation, global ranking, set difference, or anti-monotonic predicates may produce a smaller or incomparable outcome over an over-approximated state even when the DBMS behaves correctly. Therefore, SIAMT only admits

observations whose semantics preserve the approximation direction induced by the state transformation.

**Definition III.5** (Relation-Preserving Observation)**.** Given two database states $S_1$ and $S_2$ such that $S_1 \sqsubseteq_\tau S_2$, an observation $o$ is relation-preserving with respect to $\sqsubseteq_\tau$ if the observable outcomes of executing $o$ over the two states satisfy

$$\Omega(o, S_1) \preceq_o \Omega(o, S_2),$$

where $\preceq_o$ is the expected outcome relation associated with observation $o$.

The outcome relation $\preceq_o$ is determined by the abstraction representing $\Omega(o, S)$. For row-set observations, it can denote row-token or multiset containment. For count observations, it can denote cardinality monotonicity. For aggregate observations, it can denote monotonicity of aggregate summaries. For DML observations, it can denote inclusion over state-effect sets. Such a relation is valid only when the observation is deterministic and monotonic with respect to the transformed state dimension. For instance, under tuple-induced approximation, a positive count query without result truncation should satisfy count monotonicity. However, a query with `ORDER BY` and `LIMIT` may violate row containment because newly added tuples can evict previous tuples from the limited result. Similarly, anti-monotonic predicates such as `NOT EXISTS` may make previously returned tuples disappear after the state is extended. Thus, observation selection is part of the oracle design: only relation-preserving observations can be used to check state-induced approximation relations. The concrete observation-selection strategy is discussed in Section IV. We use two examples to illustrate how such observations preserve tuple-induced and admissibility-induced approximation relations.

**Example III.1** (Tuple-induced approximation)**.** Consider a state $S_1$ with a table

$$T = \begin{array}{c|c} id & v \\ \hline 1 & 10 \end{array}$$

and a transformed state $S_2$ obtained by inserting an additional tuple $(2, 20)$ into $T$:

$$T = \begin{array}{c|c} id & v \\ \hline 1 & 10 \\ 2 & 20 \end{array}$$

Thus, $S_1 \sqsubseteq_{\mathsf{tuple}} S_2$. Consider the observation

$$o : \textbf{SELECT } * \textbf{ FROM } T \textbf{ WHERE } v > 0.$$

Executing $o$ over $S_1$ returns $\{(1, 10)\}$, while executing the same observation over $S_2$ returns $\{(1, 10), (2, 20)\}$. The outcome over $S_2$ preserves the row returned over $S_1$ and may contain additional rows. Therefore, this observation preserves the tuple-induced approximation relation:

$$\Omega(o, S_1) \preceq_o \Omega(o, S_2).$$

**Example III.2** (Admissibility-induced approximation)**.** Consider two states over the same relational interface $T(id, v)$. In $S_1$, column $v$ is declared as `NOT NULL`; in $S_2$, this constraint is relaxed. Now replay the same insert workload over both states:

$$W : \textbf{INSERT INTO } T \textbf{ VALUES } (1, 10), (2, \texttt{NULL}).$$

Under $S_1$, only $(1, 10)$ is admitted because $(2, \texttt{NULL})$ violates the `NOT NULL` constraint. Under $S_2$, both tuples are admitted. Thus,

$$\mathsf{Adm}(S_1, W) = \{(1, 10)\},$$
$$\mathsf{Adm}(S_2, W) = \{(1, 10), (2, \texttt{NULL})\},$$

and therefore $S_1 \sqsubseteq_{\mathsf{adm}} S_2$. For the observation

$$o : \textbf{SELECT COUNT(*) FROM } T,$$

the outcome over $S_1$ is 1, while the outcome over $S_2$ is 2. Thus, this observation preserves the admissibility-induced approximation relation.

### D. SIAMT Oracle

Based on the above definitions, SIAMT detects *StateLogicalBug*s by checking whether the observable outcomes of relation-preserving observations respect the approximation relation induced by state transformation.

**Definition III.6** (SIAMT Oracle)**.** Given two database states $S_1$ and $S_2$, a state-induced approximation relation $S_1 \sqsubseteq_\tau S_2$, and a relation-preserving observation $o$, SIAMT reports a potential *StateLogicalBug* if

$$\Omega(o, S_1) \not\preceq_o \Omega(o, S_2).$$

Conversely, if the transformation induces the opposite relation $S_2 \sqsubseteq_\tau S_1$, SIAMT reports a potential *StateLogicalBug* if

$$\Omega(o, S_2) \not\preceq_o \Omega(o, S_1).$$

The oracle turns DBMS correctness checking into relation checking over observable outcomes. Instead of requiring an exact ground truth for each state, SIAMT relies on a known directional relation between related states and verifies whether a suitable observation preserves that direction. This abstraction allows the same testing principle to cover different manifestations of *StateLogicalBug*s, including incorrect query results under state changes, inconsistent constraint handling, plan-dependent execution errors, and erroneous DML state effects.

## IV. SchemaMorph

In this section, we present SchemaMorph, a concrete instantiation of the SIAMT principle described in Section III. As illustrated in Figure 3, SchemaMorphstarts from an initial database state constructed with schemas and initial data instances. It then derives a related mutated state through two complementary state-mutation strategies: tuple-based mutation extends the original state by appending rows and shifting data distributions, while admissibility-based mutation relaxes constraints and replays the same workload to admit additional tuples. Given the original and mutated states, SchemaMorphruns the same DQL or DML observation on both states and collects comparable query results or state

effects. The collected outcomes are then checked against the expected relation induced by the state mutation, and any violation is reported as a potential *StateLogicalBug*.

### A. Initial State Generation

SCHEMAMORPH starts each testing round by constructing an initial database state that can later be transformed and observed. The initial state contains a main table and several auxiliary tables. The main table serves as the primary target of state mutation, while the auxiliary tables are kept unchanged throughout the testing round. This design allows SCHEMAMORPH to generate join observations over multiple tables, while ensuring that only the main state component changes between the original and mutated states. For each generated table, SCHEMAMORPH records basic column information, including data types, primary keys, nullability, and indexes. Such information is used to guide later mutation and observation generation, for example, to select suitable predicate columns, generate valid join conditions, and avoid columns whose semantics are difficult to compare reliably. SCHEMAMORPH then populates the tables with concrete tuples to form a valid original state. During this process, it also prepares frequently used values and boundary values for selected columns, such as hot values, null-related values, and type-sensitive literals. These values are reused in later mutations to create distribution shifts, constraint pressure, and edge cases that are more likely to exercise different optimization and execution paths. As a result, the generated initial state provides both a valid starting point for state transformation and a controlled environment for relation-preserving observations.

### B. State Mutation

After constructing the initial state, SCHEMAMORPH transforms it into a mutated state whose observable behavior is expected to follow a predictable direction. To this end, SCHEMAMORPH uses two complementary mutation strategies: tuple-based mutation and admissibility-based mutation.

*1) Tuple-based Mutation:* For tuple-based mutation, SCHEMAMORPH takes the initial state as the original state and mutates its main table through append-only insertion. The original state is populated with a small set of seed tuples, including hot-value tuples that make baseline observations likely to return non-empty results, skewed random tuples that form a concentrated value distribution, and noise tuples that contain boundary or type-sensitive values such as `NULL`, extreme numeric values, or unusual string/date literals. These tuples provide a compact but diverse starting state for testing both common execution paths and edge-case behaviors. After collecting baseline observations on the original state, SCHEMAMORPH constructs the mutated state by appending a much larger batch of tuples to the same main table. The append operation does not delete or modify any existing tuple, so the mutated state preserves all rows in the original state and extends it with additional data. To make the mutation more effective, the newly appended tuples are biased toward a different hot-value region from the original data, creating a clear distribution shift while maintaining the containment relation between the two states. After the append operation, SCHEMAMORPH executes `ANALYZE TABLE` on the mutated table to refresh optimizer statistics, allowing the DBMS to update its cardinality estimates and potentially choose a different execution plan for the same observation. In this way, tuple-based mutation creates a larger but related mutated state that preserves the original data while increasing the chance of exposing bugs triggered by data growth, distribution shifts, type-conversion boundaries, or plan changes.

*2) Admissibility-based Mutation:* Admissibility-based mutation constructs an original state with stricter constraints and a mutated state with relaxed constraints, then replays the same insert workload so that the mutated state preserves all tuples admitted by the original state while potentially accepting additional constraint-violating tuples. Algorithm 1 summarizes this process. In Step ①, SCHEMAMORPH constructs two related schemas from the same initial schema. It samples a non-empty set of relaxation operators and uses them to derive a stricter schema for the original state and a relaxed schema for the mutated state. Although the two schemas differ in constraints, they expose the same tables and columns, so the same workload and observations can be applied to both states. SCHEMAMORPH supports the following relaxation operators:

- **`NOT NULL` relaxation.** The original state rejects `NULL` values on selected columns, while the mutated state allows such values.
- **`UNIQUE` relaxation.** The original state enforces uniqueness on selected columns, while the mutated state replaces the unique constraint with a non-unique index.
- **`CHECK` relaxation.** The original state rejects values that violate check predicates, while the mutated state removes these predicates.
- **Foreign-key relaxation.** The original state enforces referential constraints, while the mutated state removes the corresponding foreign-key checks.

In Step ②, SCHEMAMORPH generates a shared insert workload for the two states. The workload contains regular tuples to ensure that both states have enough valid baseline data, as well as constraint-stressing tuples designed for the selected relaxations. For example, it may insert `NULL` values for relaxed `NOT NULL` columns, repeated hot values for relaxed `UNIQUE` constraints, boundary or out-of-range values for relaxed `CHECK` predicates, and invalid references for relaxed foreign keys. In Step ③, SCHEMAMORPH replays the same workload on the original and mutated states and records the tuples admitted by each state. Since the mutated schema is less restrictive, its admitted tuple set is expected to contain the admitted tuple set of the original state. This replay-based construction turns schema relaxation into an observable admissibility relation between the two states. In Step ④, SCHEMAMORPH validates the generated state pair before using it for observation. It first checks whether the mutated state admits sufficiently more tuples than the original state; otherwise, the relaxation may be too weak to expose

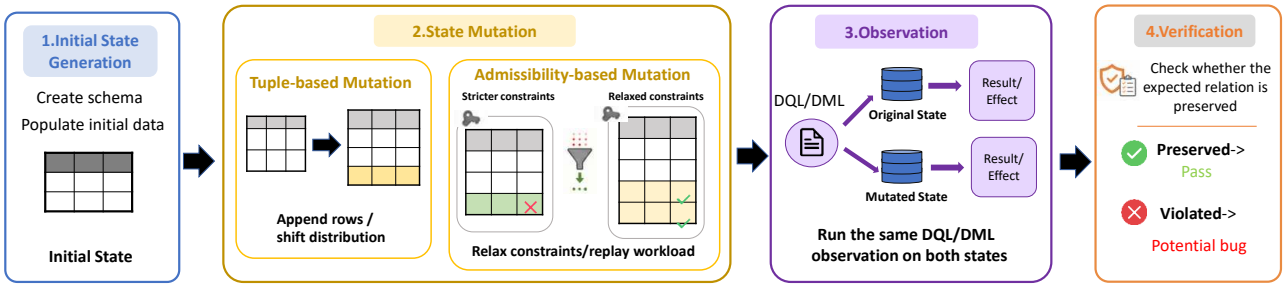


Fig. 3: **Overview of SCHEMAMORPH**

**Algorithm 1** Admissibility-based state mutation

**Require:** Initial schema $\Sigma$, auxiliary tables $\mathcal{A}$, relaxation operators $\mathcal{R}$
**Ensure:** Original state $S_{orig}$ and mutated state $S_{mut}$

**// Step ①: Construct related schemas**
Sample a non-empty set of relaxation operators $\mathcal{R}' \subseteq \mathcal{R}$.
Construct a stricter schema $\Sigma_{orig}$ by enforcing the constraints in $\mathcal{R}'$.
Construct a relaxed schema $\Sigma_{mut}$ by removing or weakening these constraints.
Create two main tables with the same relational interface under $\Sigma_{orig}$ and $\Sigma_{mut}$.
**// Step ②: Generate a shared modification workload**
Initialize an insert workload $W \leftarrow \emptyset$.
Add regular tuples to $W$ to ensure both states contain valid baseline data.
**for all** relaxation operator $r \in \mathcal{R}'$ **do**
    Add constraint-stressing tuples for $r$ to $W$.
**end for**
**// Step ③: Replay workload on both states**
Replay $W$ on $\Sigma_{orig}$ and record accepted tuples $\mathsf{Adm}_{orig}$.
Replay $W$ on $\Sigma_{mut}$ and record accepted tuples $\mathsf{Adm}_{mut}$.
**// Step ④: Validate the generated state pair**
**if** $|\mathsf{Adm}_{mut}| - |\mathsf{Adm}_{orig}|$ is too small **then**
    **return** DISCARD
**end if**
**if** $\mathsf{Adm}_{orig} \not\subseteq \mathsf{Adm}_{mut}$ **then**
    **return** DISCARD
**end if**
Build $S_{orig}$ from $\Sigma_{orig}$, $\mathsf{Adm}_{orig}$, and metadata.
Build $S_{mut}$ from $\Sigma_{mut}$, $\mathsf{Adm}_{mut}$, and metadata.
**return** $(S_{orig}, S_{mut})$.

meaningful behavioral differences. It also checks whether all tuples admitted by the original state are preserved in the mutated state. If either check fails, the state pair is discarded. Otherwise, SCHEMAMORPH returns the original and mutated states for subsequent observation and verification.

**Optimization Summary.** The two mutation strategies above define the basic state relations used by SCHEMAMORPH. To address Challenge #1, we further apply several lightweight optimizations to avoid weak state pairs and increase the chance of exercising different DBMS reasoning paths.

**O1: Data-distribution shaping.** Mainly for tuple-based mutation, SCHEMAMORPH uses hot values, boundary values, and shifted value distributions to make the mutated state differ from the original state in a controlled, observable way.

**O2: Index and statistics manipulation.** For tuple-based mutation, SCHEMAMORPH refreshes statistics after data expansion; for admissibility-based mutation, relaxed constraints may also change index properties, such as unique versus non-unique indexes.

### C. Observation

After constructing the original and mutated states, SCHEMAMORPH applies the same observation to both states. The purpose of observation is to expose whether the approximation relation induced by state mutation is preserved during query execution or data modification. Depending on the mutation strategy, the two outcomes are obtained in slightly different ways. For tuple-based mutation, SCHEMAMORPH executes the observation before and after append-only mutation on the same main table. For admissibility-based mutation, SCHEMAMORPH executes the observation on two related tables with the same relational interface, replacing only the physical table name while keeping the logical statement unchanged. In both cases, the observation is designed to be deterministic and relation-preserving, so that any violation of the expected relation can be used as a reliable bug signal. To reduce ineffective executions, SCHEMAMORPH further uses `EXPLAIN` to inspect candidate observations before executing them when supported by the target DBMS. Observations whose plans remain unchanged across the original and mutated states are deprioritized, while those that trigger plan changes are executed first because they are more likely to expose plan-dependent state-reasoning bugs.

*1) DQL Observation:* For DQL observation, SCHEMAMORPH generates deterministic queries whose results are expected to preserve the approximation relation induced by state mutation. These queries are constructed over the main table and auxiliary tables, covering common query forms such as single-table selections, inner joins, self-joins, derived-table queries, `IN`/`EXISTS` subqueries, and `UNION ALL` queries. To avoid unsound observations, SCHEMAMORPH excludes query constructs whose semantics may legally break monotonicity, such as `LIMIT`, anti-joins, set difference, and other forms that may remove previously returned rows after the state is extended. Thus, a DQL observation is expected to preserve the corresponding query outcome relation between the original and mutated states. For admissibility-based mutation, SCHEMAMORPH further biases DQL generation toward relaxed columns and constraints. For example, queries may refer to nullable columns, columns with relaxed uniqueness, or columns involved in relaxed constraints. This makes the observation more likely to exercise the state differences introduced by schema relaxation, rather than querying columns unrelated to the mutation.

*2) DML Observation:* In addition to DQL queries, SCHEMAMORPH uses DML statements as observations to check whether state effects preserve the expected relation. Unlike DQL observations, which compare returned rows, DML observations compare the rows affected by data-modification operations. Under a state-induced over-approximation relation, if the mutated state preserves the relevant rows from the original state, then a suitable DML statement should also preserve the corresponding affected-row set. Therefore, SCHEMAMORPH generates deterministic `DELETE` and `UPDATE` statements with monotonic predicates, so that rows affected in the original state are expected to remain affected in the mutated state. To observe DML effects without permanently changing the database, SCHEMAMORPH executes each DML statement inside a transaction. It records a pre-execution snapshot of the target table, executes the DML statement, records a post-execution snapshot, and computes their difference as the state effect. Finally, SCHEMAMORPH rolls back the transaction, so the observation does not affect subsequent tests. For `DELETE`, the effect is the set of primary keys that disappear after execution; for `UPDATE`, the effect is the set of primary keys whose row digests change. SCHEMAMORPH does not use `INSERT` as a DML observation, because the mutated state may contain more rows and thus have different primary-key conflict behavior, making the direction of successful insertions difficult to predict.

*3) Observable Outcome Extraction:* After executing an observation, SCHEMAMORPH extracts an abstract outcome that can be compared across the original and mutated states. For DQL observations, SCHEMAMORPH records two kinds of outcomes. First, when the query result is small enough to materialize, SCHEMAMORPH normalizes each returned row into a row digest and records the multiset of row digests. This supports set-level comparison, which checks whether rows observed in the original state are still preserved in the mutated state. Second, SCHEMAMORPH computes lightweight summary values over the query result, such as the number of returned rows and selected numeric summaries. These summaries provide value-level evidence of whether the mutated state loses or changes values that should be preserved, without requiring full row-by-row comparison. For DML observations, the final outcome is the abstract state effect of the statement. The pre- and post-execution table snapshots are used only as intermediate data to compute this effect. SCHEMAMORPH records the final DML outcome as the set of affected primary keys: for `DELETE`, these are the primary keys that disappear after execution; for `UPDATE`, these are the primary keys whose row digests change.

**Optimization Summary.** The observation stage determines whether the constructed state relation can be effectively exposed. SCHEMAMORPH applies two lightweight optimizations to select observations that are both relation-preserving and likely to exercise different DBMS behaviors.

**O3: Plan-change-aware prioritization.** SCHEMAMORPH uses query plan feedback to prioritize observations that trigger different plans across the original and mutated states.

**O4: Relaxation-aware observation biasing.** For admissibility-based mutation, SCHEMAMORPH further biases observations toward relaxed columns and constraints, making the induced admissibility difference more likely to be exercised.

### D. Verification

After extracting observable outcomes from the original and mutated states, SCHEMAMORPH checks whether they satisfy the relation induced by the corresponding state mutation. For DQL observations, SCHEMAMORPH compares the extracted outcomes from two perspectives: set-level preservation and value-level consistency. Set-level verification checks whether the row digests observed in the original state are still contained in the mutated state. Value-level verification checks whether numerical summaries follow the expected direction; for example, under tuple-induced over-approximation, the count returned over the mutated state should be no smaller than the count returned over the original state. For DML observations, SCHEMAMORPH compares the affected-row sets extracted from the two states; if a row is affected in the original state, it should also be affected in the mutated state. Any violation of these outcome relations is reported as a potential bug of the state-induced approximation relation.

### E. Additional Implementation Details

We implemented SCHEMAMORPH as a prototype DBMS testing system in Python, consisting of 36,132 lines of code. For observation generation, SCHEMAMORPH builds on VALSCOPE [21] and DMLSCOPE [33], reusing their query and DML generation infrastructure while retaining only operators and query forms that satisfy the relation-preserving requirements of SIAMT. To improve the usability of detected bugs, we adopt SQLESS [34] to simplify failing cases and isolate their root causes, and further apply a deduplication strategy following PINOLO [19] to eliminate redundant bug reports. SCHEMAMORPH is primarily designed for MySQL-compatible systems, including PolarDB and MariaDB, but its modular architecture allows for straightforward adaptation to other DBMS dialects by customizing syntax- and type-related components. In practice, such adaptation requires only lightweight modifications, and can be further automated using tools such as QTRAN [35], which facilitates cross-dialect deployment of MT techniques.

## V. EVALUATION

### A. Experimental Setup

**Tested DBMS.** We evaluate SCHEMAMORPH on five widely used DBMSs, as summarized in Table II. These DBMSs were chosen based on their popularity and extensive use in real-world applications. All of these DBMSs have undergone extensive testing [11]–[13], [15]–[20], [32], [35]. Although finding new bugs in such widely tested systems

is challenging, it serves to validate the effectiveness of SCHEMAMORPH. In our experiments, SCHEMAMORPH tests the latest versions of each DBMS. We run SCHEMAMORPH on each DBMS for 24 hours.

TABLE II: **The demographics of the DBMSs under test**

| DBMS | Version | GitHub Stars | DB-Engine | First Release |
|---|---|---|---|---|
| MySQL [36] | 9.5.0 | 11.9K | 2 | 1995 |
| MariaDB [37] | 12.2.2 | 6.7K | 13 | 2009 |
| Percona [38] | 8.4.8-8 | 1.2K | 122 | 2008 |
| PolarDB [39] | 8.0.32 | 1.6K | 105 | 2017 |
| OceanBase [40] | 5.7.25 | 9.8K | 99 | 2010 |

**Environment**. We conduct the experiments on one server with 64-cores Intel(R) Xeon(R) Platinum 8358P @ 2.60GHz and 500 GB memory. The server operates under the Ubuntu 20.04 OS, with the 5.4.0-135-generic Linux kernel.

### *B. Bug Detection*

Table III shows the status of the logical bugs detected by SCHEMAMORPH across different DBMSs. In total, we reported 21 previously unknown logical bugs in five DBMSs, including 8 in MySQL, 2 in MariaDB, 7 in Percona, 3 in PolarDB, and 1 in OceanBase. All of these bugs have been confirmed by developers. All reported bugs are state-related: they are triggered by changes in database states. These results demonstrate that SCHEMAMORPH can effectively uncover hidden logical inconsistencies caused by incorrect handling of database-state information in mature and widely used DBMSs.

**Bug Importance.** Although anecdotal, developer feedback is an important indicator of an approach's effectiveness and the found bugs' importance. Two DBMS companies reached out to us about our testing efforts; both were interested in how we found the bugs, and one invited us to help them find more bugs. For example, a developer from PolarDB commented: "Outstanding work, this tool looks very useful, and we will fully support any technical requirements you may need." Additionally, we received positive feedback on public bug tracking platforms. Among the confirmed bugs, 7 in MySQL were labeled as a serious bug, indicating it is a high-impact and difficult-to-resolve issue; furthermore, in MariaDB, one bug was labeled as "Major", which represents the highest severity level in the system.

**Triggering Strategies.** Among the 21 reported bugs, 17 were triggered by tuple-based mutation, where append-only data expansion, distribution shifts, and statistics refresh exposed incorrect results under changed database states. The remaining 4 bugs were triggered by admissibility-based mutation, where constraint relaxation and workload replay revealed incorrect handling of schema constraints and admitted tuples. These results suggest that tuple-induced approximation is effective for plan- and data-distribution-dependent bugs, while admissibility-induced approximation provides additional coverage for constraint- and schema-data-related bugs.

**Temporal Analysis.** We further analyze the earliest version-introduction year of the bugs to understand whether SCHEMAMORPH can uncover long-standing logical bugs. As shown in Table III, many confirmed bugs had already existed for several years before being reported by SCHEMAMORPH. Among the 18 confirmed bugs, 15 were introduced before 2023, and 2 can be traced back to versions earlier than 2020. The oldest confirmed bug was introduced in 2016, indicating that some state-related logical bugs can remain hidden in mature DBMSs for nearly a decade. Given that these DBMSs have been widely used and repeatedly tested by prior DBMS testing tools, these long-lived bugs suggest that existing approaches may still miss state-related logical inconsistencies.

TABLE III: **Status and triggering strategies of detected logical bugs.**

| ID | Bug ID | Status | Tuple | Adm. | Year |
|---|---|---|---|---|---|
| 1 | MySQL#120158 | Confirmed | ● | ❍ | 2016 |
| 2 | MySQL#120253 | Confirmed | ● | ❍ | 2021 |
| 3 | MySQL#120275 | Confirmed | ● | ❍ | 2021 |
| 4 | MySQL#120284 | Confirmed | ● | ❍ | 2020 |
| 5 | MySQL#120285 | Confirmed | ● | ❍ | 2020 |
| 6 | MySQL#120296 | Confirmed | ● | ❍ | 2021 |
| 7 | MySQL#120524 | Confirmed | ❍ | ● | 2021 |
| 8 | MySQL#120525 | Confirmed | ❍ | ● | 2021 |
| 9 | MariaDB#MDEV-39553 | Confirmed | ● | ❍ | 2017 |
| 10 | MariaDB#MDEV-39717 | Confirmed | ● | ❍ | 2016 |
| 11 | Percona#PS-11024 | Confirmed | ● | ❍ | 2021 |
| 12 | Percona#PS-11025 | Confirmed | ● | ❍ | 2022 |
| 13 | Percona#PS-11144 | Confirmed | ● | ❍ | 2022 |
| 14 | Percona#PS-11147 | Confirmed | ● | ❍ | 2021 |
| 15 | Percona#PS-11148 | Confirmed | ● | ❍ | 2022 |
| 16 | Percona#PS-11187 | Confirmed | ❍ | ● | 2022 |
| 17 | Percona#PS-11188 | Confirmed | ❍ | ● | 2022 |
| 18 | PolarDB#283 | Confirmed | ● | ❍ | 2024 |
| 19 | PolarDB#284 | Confirmed | ● | ❍ | 2024 |
| 20 | PolarDB#285 | Confirmed | ● | ❍ | 2024 |
| 21 | OceanBase#2401 | Confirmed | ● | ❍ | 2025 |

**Note.** "Tuple" and "Adm." indicate whether the bug was triggered by tuple-based mutation or admissibility-based mutation, respectively. "Year" denotes the earliest version-introduction year identified during bug analysis.

**Case Study.** Figure 1 has shown a tuple-based bug triggered by append-only state expansion. Due to space limitations, we provide another detailed case study of an admissibility-based MySQL bug in our released GitHub repository [41].

### *C. Comparative Study*

To further evaluate the effectiveness of SCHEMAMORPH, we compare it with existing DBMS logical bug detection approaches from two perspectives: tool level and oracle level. At the tool level, we compare SCHEMAMORPH with eight representative testing approaches, including NOREC [11], TLP [15], RADAR [8], DDLCHECK [9], PINOLO [19], EET [18], VALSCOPE [21], and DQP [42]. Since the baseline tools do not support all DBMSs tested by SCHEMAMORPH, we restrict each comparison to the DBMSs commonly supported by SCHEMAMORPH and the corresponding baseline. As shown in Table IV, on the commonly supported DBMSs, SCHEMAMORPH finds 3, 4, 10, 8, 8, 5, 12, and 9 more bugs than NOREC, TLP, RADAR, DDLCHECK, PINOLO, EET, VALSCOPE, and DQP, respectively. These results show that SCHEMAMORPH can uncover state-related logical bugs that remain undetected by existing DBMS testing tools.

At the oracle level, we examine whether existing test oracles can reproduce the bugs found by SCHEMAMORPH. We manually applied the oracles of representative baselines

TABLE IV: **Number of logical bugs triggered in 24 hours.**

| DBMS | NoREC | TLP | Radar | DDLCheck | Pinolo | EET | ValScope | DQP | Ours |
|---|---|---|---|---|---|---|---|---|---|
| MySQL | - | 4 | 0 | 2 | 2 | 3 | 2 | 1 | 8 |
| MariaDB | 0 | - | 0 | 0 | 1 | - | - | 0 | 2 |
| Percona | - | - | - | - | - | - | 2 | - | 7 |
| PolarDB | - | - | - | - | - | - | 2 | - | 3 |
| OceanBase | 0 | - | - | - | 0 | - | - | - | 1 |
| **Total** | 0 | 4 | 0 | 2 | 3 | 3 | 6 | 1 | **21** |
| **Increment** | 3 | 4 | 10 | 8 | 8 | 5 | 12 | 9 | |

**Note.** "-" indicates that the tool does not support the DBMS. "Increment" is computed on the commonly supported DBMSs between each baseline and SCHEMAMORPH.

TABLE V: **Ablation of optimization strategies on MySQL.**

| Variant | Unique Query Plans | Bugs |
|---|---|---|
| SCHEMAMORPH | 267,658 | 8 |
| SCHEMAMORPH w/o O1 | 250,154 | 5 |
| SCHEMAMORPH w/o O2 | 265,795 | 6 |
| SCHEMAMORPH w/o O3 | 198,516 | 2 |
| SCHEMAMORPH w/o O4 | 216,849 | 5 |

**Note.** O1 denotes data-distribution shaping, O2 denotes index and statistics manipulation, O3 denotes plan-change-aware prioritization, and O4 denotes relaxation-aware observation biasing.

to our detected bugs and found that none of them could report the corresponding violations. The main reason is that existing oracles mainly check query-level equivalence, query-level approximation, or state equivalence, while our bugs require reasoning about directional approximation relations induced by database-state transformations. This shows that SCHEMAMORPH explores a different testing space that is not covered by existing query-level or equivalence-based oracles.

### *D. Ablation Study*

To evaluate the contribution of each optimization strategy, we conduct an ablation study on MySQL. We compare the full version of SCHEMAMORPH with four variants, each disabling one optimization strategy introduced in Section IV. Specifically, we disable data-distribution shaping (O1), index and statistics manipulation (O2), plan-change-aware prioritization (O3), and relaxation-aware observation biasing (O4), respectively. We measure two metrics: the number of unique query plans exercised during testing and the number of logical bugs triggered within the same time budget. The number of unique query plans reflects the diversity of DBMS execution behaviors explored by each variant, while the number of bugs directly measures bug-triggering effectiveness. As shown in Table V, the full version of SCHEMAMORPH achieves the best overall performance, exercising 267,658 unique query plans and triggering 8 bugs. Removing O3 leads to the largest drop, reducing the number of unique query plans to 198,516 and the number of bugs to 2, which shows the importance of plan-change-aware prioritization in filtering ineffective observations. Removing O1, O2, and O4 also reduces the number of triggered bugs to 5, 6, and 5, respectively, indicating that distribution shaping, index/statistics manipulation, and relaxation-aware observation biasing all contribute to bug detection.

## VI. RELATED WORK

**Logical Bug Detection in DBMSs.** A variety of approaches have been proposed to detect logical bugs in DBMSs [8], [11], [13], [15]–[20], [42], [43]. For example, VALSCOPE [21] further explores approximation-based testing from the perspective of value semantics, checking whether transformed queries preserve expected value-level relations. Other approaches focus on different sources of logical bugs, such as expression evaluation [18], [20], constant folding and propagation [32], and metadata- or schema-related behaviors [8], [9]. As discussed in Section I and Section II, existing DBMS testing techniques either induce metamorphic relations by transforming queries, or transform database states under equivalence-based oracles. In contrast, SCHEMAMORPH induces approximation relations directly from database-state transformations. This design extends the oracle space of existing DBMS testing techniques and enables SCHEMAMORPH to expose *StateLogicalBug*s caused by incorrect reasoning about data distributions, constraints, metadata, and execution-plan changes.

**DBMS Test-Case Generation.** A variety of approaches have been proposed for generating diverse test cases for DBMSs, with the aim of improving coverage and revealing potential bugs. These techniques [35], [44]–[47] typically focus on generating valid and varied SQL queries, but do not specifically target logical bugs. For example, SQLSMITH [44] is a grammar-based DBMS fuzzer that embeds SQL grammar rules to generate complex SQL queries. In addition to general query generation, some approaches [48], [49] have been specifically designed to aid in bug detection. For example, SQLRIGHT [48] leverages code coverage feedback to enhance test-case generation. Unlike these works, SCHEMAMORPH does not target general-purpose SQL generation or coverage improvement. Instead, it constructs related database states with known approximation relations and applies relation-preserving DQL/DML observations to check expected containment or monotonicity. Thus, SCHEMAMORPH complements existing test-case generation techniques: prior generators provide diverse SQL inputs, while SCHEMAMORPH provides a state-induced approximation oracle for detecting *StateLogicalBug*s.

## VII. CONCLUSION

In this paper, we proposed *State-Induced Approximate Metamorphic Testing* (SIAMT) for detecting state-related logical bugs in DBMSs. We implemented SCHEMAMORPH, which constructs related database states and checks whether DQL/DML observations preserve the induced approximation relations. SCHEMAMORPH found 21 previously unknown *StateLogicalBug*s across five widely used DBMSs, all of which were confirmed by developers. We release the source code of SCHEMAMORPH at [50].


## ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. This work is supported by the Xinmiao Project of ZheJiang Provincial Applied Basic Research Program (Grant No. 2026XMZD032), the Zhejiang Pioneer (Jianbing) Project (2025C01198), the National Science Foundation of China (No. 62372398, No. 72342025), Zhejiang Provincial Science Foundation of China (No. LQK26F020002), and the Fundamental Research Funds for the Central Universities (No. 226-2025-00067).

## REFERENCES


[1] D. Florescu, A. Levy, and A. Mendelzon, "Database techniques for the world-wide web: A survey," *ACM Sigmod Record*, vol. 27, no. 3, pp. 59–74, 1998.

[2] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets artificial intelligence: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 3, pp. 1096–1116, 2020.

[3] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, 1974, pp. 249–264.

[4] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM computing surveys (CSUR)*, vol. 18, no. 4, pp. 323–364, 1986.

[5] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *The VLDB Journal*, vol. 22, no. 1, pp. 73–98, 2013.

[6] P. Godfrey, J. Gryz, and C. Zuzarte, "Exploiting constraint-like data characterizations in query optimization," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, 2001, pp. 582–592.

[7] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis, "Complete yet practical search for minimal query reformulations under constraints," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1015–1026.

[8] J. Song, W. Dou, Y. Gao, Z. Cui, Y. Zheng, D. Wang, W. Wang, J. Wei, and T. Huang, "Detecting metadata-related logic bugs in database systems via raw database construction," *Proceedings of the VLDB Endowment*, vol. 17, no. 8, pp. 1884–1897, 2024.

[9] J. Song, W. Dou, Y. Zheng, Y. Gao, Z. Cui, W. Wang, and J. Wei, "Detecting schema-related logic bugs in relational dbmss via equivalent database construction," *Proceedings of the VLDB Endowment (VLDB)*, vol. 18, no. 7, pp. 2281–2294, 2025.

[10] J. Liang, Z. Wu, J. Fu, M. Wang, C. Sun, and Y. Jiang, "Mozi: Discovering dbms bugs via configuration-based equivalent transformation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[11] M. Rigger and Z. Su, "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*, 2020, pp. 1140–1152.

[12] M. Kamm, M. Rigger, C. Zhang, and Z. Su, "Testing graph database engines via query partitioning," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 23)*, 2023, pp. 140–149.

[13] J. Song, W. Dou, Z. Cui, Q. Dai, W. Wang, J. Wei, H. Zhong, and T. Huang, "Testing database systems via differential query execution," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE 23)*. IEEE, 2023, pp. 2072–2084.

[14] Y F, "Bug #120253: Index Lookup Skips CAST for LONGTEXT-to-DATE Coercion in JOIN, Causing Plan-Dependent Result Divergence," MySQL Bugs. Available at: https://bugs.mysql.com/bug.php?id=120253, 2026, accessed: 2026-05-05.

[15] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[16] ——, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 667–682.

[17] X. Tang, S. Wu, D. Zhang, F. Li, and G. Chen, "Detecting logic bugs of join optimizations in dbms," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[18] Z.-M. Jiang and Z. Su, "Detecting logic bugs in database engines via equivalent expression transformation," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 821–835.

[19] Z. Hao, Q. Huang, C. Wang, J. Wang, Y. Zhang, R. Wu, and C. Zhang, "Pinolo: Detecting logical bugs in database management systems with approximate query synthesis," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 345–358.

[20] W. Deng, J. Liang, Z. Wu, J. Fu, and Y. Jiang, "Detecting logic bugs in dbmss via equivalent data construction," *Proceedings of the ACM on Management of Data*, vol. 3, no. 6, pp. 1–25, 2025.

[21] L. Lin, L. Chen, and R. Wu, "ValScope: Value-semantics-aware metamorphic testing for detecting logical bugs in DBMSs," in *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2026.

[22] W. E. Howden, "Theoretical and empirical studies of program testing," *IEEE Transactions on Software Engineering*, no. 4, pp. 293–298, 2006.

[23] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB*, vol. 98, 1998, pp. 618–622.

[24] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 1243–1251.

[25] Wikipedia, "Databases," https://en.wikipedia.org/wiki/Database, 2026, accessed: 2026-01-23.

[26] R. F. Van Der Lans, *The SQL Standard: A Complete Guide Reference*. Prentice Hall International (UK) Ltd., 1989.

[27] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Pearson, 2016.

[28] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[29] Z. Gu, M. A. Soliman, and F. M. Waas, "Testing the accuracy of query optimizers," in *Proceedings of the Fifth International Workshop on Testing Database Systems*, 2012, pp. 1–6.

[30] V. Poosala, *Histogram-based estimation techniques in database systems*. The University of Wisconsin-Madison, 1997.

[31] G. Graefe, "Modern b-tree techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[32] C. Zhang and M. Rigger, "Constant optimization driven database system testing," *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, pp. 1–24, 2025.

[33] "DMLSCOPE: Artifact repository," https://anonymous.4open.science/r/DMLScope-9F03/, 2026, accessed: 2026-05-28.

[34] L. Lin, Z. Hao, C. Wang, Z. Wang, R. Wu, and G. Fan, "Sqless: Dialect-agnostic sql query simplification," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 24)*, 2024, pp. 743–754.

[35] L. Lin, Q. Zhu, H. Chen, Z. Wang, R. Wu, and X. Xie, "Qtran: Extending metamorphic-oracle based logical bug detection techniques for multiple-dbms dialect support," in *Proceedings of the 34nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 25)*, 2025, pp. 731–752.

[36] MySQL, "Mysql," 2023, accessed Dec 2025. [Online]. Available: https://www.mysql.com/

[37] M. Foundation, "Mariadb," 2023, accessed Dec 2025. [Online]. Available: https://mariadb.org/

[38] Percona, "Percona - open source database software and services," 2023, accessed Dec 2025. [Online]. Available: https://www.percona.com/

[39] A. Cloud, "Polardb product overview," https://help.aliyun.com/zh/polardb/product-overview/, 2023, accessed Dec 2025.

[40] OceanBase, "Oceanbase open source," 2023, accessed Dec 2025. [Online]. Available: https://www.oceanbase.com/product/opensource

[41] SchemaMorph Developers, "Case Study: A MySQL Bug Triggered by Admissibility-Based Mutation," https://github.com/SchemaMorph/SchemaMorph/blob/main/Case%20Study%20A%20MySQL%20Bug%20Triggered%20by%20Admissibility-Based%20Mutation.md, 2026, accessed: May 30, 2026.

[42] J. Ba and M. Rigger, "Keep it simple: Testing databases via differential query plans," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.

[43] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su, "Detecting transactional bugs in database engines via {graph-based} oracle construction," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 397–417.

[44] Anse1, "Sqlsmith: A random sql query generator," https://github.com/anse1/sqlsmith, 2020., accessed Dec 2025.

[45] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing database management systems with language validity and coverage feedback," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 955–970.

[46] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free dbms fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 22)*, 2022, pp. 1–12.

[47] Z.-M. Jiang, J.-J. Bai, and Z. Su, "{DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4949–4965.

[48] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of {DBMS} with coverage-based guidance," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4309–4326.

[49] J. Ba and M. Rigger, "Testing database engines via query plan guidance," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE 23)*. IEEE, 2023, pp. 2060–2071.

[50] L. Lin, Y. Fei, L. Bao, R. Wu, and D. Zhang, "SchemaMorph: State-induced approximate metamorphic testing for detecting logical bugs in dbmss," https://github.com/SchemaMorph/SchemaMorph, 2026, software repository.