Detecting Isolation Anomalies in Relational DBMSs

RUI YANG[*], Institute of Software Chinese Academy of Sciences, China
ZIYU CUI[*], Institute of Software Chinese Academy of Sciences, China
WENSHENG DOU[*†‡], Institute of Software Chinese Academy of Sciences, China
YU GAO[*], Institute of Software Chinese Academy of Sciences, China
JIANSEN SONG[*], Institute of Software Chinese Academy of Sciences, China
XUDONG XIE[*], Institute of Software Chinese Academy of Sciences, China
JUN WEI[*†], Institute of Software Chinese Academy of Sciences, China

Relational Database Management Systems (DBMSs) utilize transactions to ensure data consistency and integrity, while providing multiple isolation levels to strike a balance between consistency and performance. However, isolation anomalies in relational DBMSs can undermine their claimed isolation levels, and lead to severe consequences, e.g., incorrect query results and database states. Existing isolation checkers can only work on simple *key-value*-like data models and the associated $read(key)$ and $write(key, value)$ operations. Therefore, they cannot be directly applied to relational DBMSs that support relational data models and complex SQL operations.

In this paper, we propose a novel black-box *Isolation checker for Relational DBMSs*, `IsoRel`, which can support relational data models and complex SQL operations. To infer dependencies among transactions in relational DBMSs, we first design an isolation-agnostic SQL statement instrumentation approach to record the data rows accessed by each SQL statement by utilizing two auxiliary columns in each database table. We then utilize the recorded data rows of each SQL statement to construct a transaction dependency graph for relational transactions, and identify isolation anomalies based on anomaly patterns. We evaluate `IsoRel` on five widely-used relational DBMSs, i.e., MySQL, PostgreSQL, MariaDB, CockroachDB, and TiDB, and all their supported isolation levels. Our evaluation reveals a total of 48 unique isolation anomalies that violate the isolation levels defined by Adya.

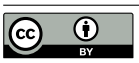
CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Database transaction, isolation level, isolation anomaly

ACM Reference Format:
Rui Yang, Ziyu Cui, Wensheng Dou, Yu Gao, Jiansen Song, Xudong Xie, and Jun Wei. 2025. Detecting Isolation Anomalies in Relational DBMSs. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA076 (July 2025), 23 pages. https://doi.org/10.1145/3728953

[*]Affiliated with Key Lab of System Software at Chinese Academy of Sciences, State Key Lab of Computer Science at Institute of Software Chinese Academy of Sciences, and University of Chinese Academy of Sciences, Beijing.
[†]Affiliated with Nanjing Institute of Software Technology, University of Chinese Academy of Sciences, Nanjing.
[‡]Wensheng Dou is the corresponding author.

Authors' Contact Information: Rui Yang, yangrui22@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Ziyu Cui, cuiziyu20@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Wensheng Dou, wsdou@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Yu Gao, gaoyu15@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Jiansen Song, songjiansen20@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Xudong Xie, xiexudong23@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Jun Wei, wj@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China.



ACM 2994-970X/2025/7-ARTISSTA076
https://doi.org/10.1145/3728953

1 Introduction

Relational Database Management Systems (DBMSs), e.g., MySQL [8], PostgreSQL [10], MariaDB [7], CockroachDB [3], and TiDB [15], have been widely used in many applications as data storage infrastructure. They utilize the relational data model [31] to store and manage data, which organizes data in two-dimensional tables consisting of rows and columns, and supports many complex features, e.g., foreign keys and indexes. Relational DBMSs provide Structured Query Language (SQL) [29] as the standard query language to retrieve and manipulate data. SQL supports a variety of advanced functionalities, e.g., range queries, compound queries, subqueries, and aggregate functions.

Relational DBMSs utilize transactions to ensure data consistency and integrity [22, 47]. A transaction usually consists of a group of SQL statements that are executed atomically in relational DBMSs. Transactions are characterized by ACID properties, i.e., Atomicity, Consistency, Isolation, and Durability [36, 37, 39]. Among these properties, isolation is particularly important because it ensures that concurrent transactions operate independently of one another, preventing the execution of one transaction from being interfered with by others.

However, stronger isolation can lead to greater degradation in performance in relational DBMSs. To strike a balance between consistency and performance, relational DBMSs typically provide several isolation levels, e.g., `Read Uncommitted (RU)`, `Read Committed (RC)`, `Snapshot Isolation (SI)`, `Repeatable Read (RR)`, and `Serializable (SER)` [2, 16, 17, 19–21, 27]. Each isolation level specifies how concurrent transactions are isolated from one another. For example, under `Read Committed`, a query in a transaction can read the data committed by other transactions. In contrast, under `Repeatable Read`, a query in a transaction can only read the data committed by other transactions before the snapshot of its own transaction was created.

Unfortunately, incorrect implementations of transaction mechanisms can undermine the isolation levels claimed by relational DBMSs, leading to *isolation anomalies*. Isolation anomalies can lead to serious consequences, e.g., incorrect query results and incorrect database states. In relational DBMSs, isolation anomalies usually involve relational data models and complex SQL operations. A recent study on 140 real-world transaction bugs collected from six relational DBMSs has found that 31.4% of these transaction bugs are caused by isolation anomalies, and 93.2%[1] of these isolation anomalies involve relational data models or complex SQL operations [33].

Existing isolation checkers [24, 27, 28, 30, 42, 46, 49, 58, 62] mainly work on *key-value*-like data models, in which data is organized in a *key-value* format. These checkers analyze the execution history of transactions based on $read(key)$ and $write(key, value)$ operations, infer dependencies among transactions based on the read and written *key*s, and construct a transaction dependency graph to detect isolation anomalies. These checkers are limited to detecting isolation anomalies within *key-value*-like data models and the associated $read(key)$ and $write(key, value)$ operations.

Consequently, existing isolation checkers designed for *key-value*-like data models cannot be easily adapted to relational DBMSs. In *key-value*-like data models, transaction dependencies can be easily inferred from the keys accessed by transaction operations (i.e., $read(key)$ and $write(key, value)$). In contrast, relational DBMSs lack a default *key-value* structure within their tables and support complex SQL operations (e.g., predicates, subqueries, and compound queries). As a result, the dependencies among transactions become less apparent, complicating the task of determining how transactions interact with one another. Specifically, the following two features significantly hinder the detection of isolation anomalies in relational DBMSs.

[1]Few isolation anomalies in relational DBMSs can be triggered by simple key-$value$-like data models. For example, we can construct a simple table t with two columns $c1$ and $c2$ ($c1$ for key and $c2$ for $value$). We can further use simple SQL operations to simulate $read(key)$ and $write(key, value)$, e.g., SELECT c2 FROM t WHERE c1 = "key". We do not think these isolation anomalies involve complex SQL operations in relational DBMSs.

- **Relational data models.** Unlike *key-value* data models that utilize *key* and *value* to manage data, the data models in relational DBMSs [31] manage data in tables and support complex data structures (e.g., multiple tables, multi-column primary keys, and foreign keys), and column constraints (e.g., unique constraints and data constraints). A relational table may contain no keys, compound keys, etc. Existing isolation checkers are designed specifically for *key-value*-like data models [24, 27, 28, 30, 42, 46, 49, 58, 62], and are not equipped to handle the complexities of relational data models in relational DBMSs.
- **Flexible SQL operations.** Relational DBMSs support different kinds of complex SQL operations, e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `REPLACE`. Unlike the simple $read(key)$ and $write(key, value)$ operations, SQL operations can support predicates, subqueries, compound queries, aggregate functions, etc. For example, a `SELECT` statement "`SELECT name, gender FROM student WHERE age < 20`" retrieves the name and gender of students who are under 20. An `UPDATE` statement "`UPDATE student SET age = age+1 WHERE age < 20`" updates the age by adding 1 for students who are under 20. Existing isolation checkers [24, 27, 28, 30, 42, 46, 49, 58, 62] are limited to supporting simple read and write operations based on *key*, i.e., $read(key)$ and $write(key, value)$. Thus, they cannot support such complex SQL operations in relational DBMSs.

In this paper, we propose a novel black-box *Isolation checker for Relational DBMSs*, `IsoRel`. The core idea of `IsoRel` is to add two auxiliary columns into each table and instrument SQL statements in an isolation-agnostic manner. By analyzing the execution history of the instrumented SQL statements, `IsoRel` can infer dependencies among transactions that involve relational data models and complex SQL operations. Specifically, we add an auxiliary column *rowId* into each table and assign a unique ID to each row in the table. We further add an auxiliary column *wList* into each table, which is used to maintain a list of transactions that create or modify a row. Based on these two auxiliary columns, we instrument each read operation to additionally retrieve the *rowId*s and *wList*s of their retrieved rows, and instrument each write operation to update *wList*s for their modified rows. Our auxiliary columns and SQL statement instrumentation are isolation-agnostic[2], and do not affect the concurrent execution of the original transactions. Then, we infer the dependencies among transactions based on the *rowId*s and *wList*s of the rows accessed in each SQL statement, and construct the dependency graph among transactions. Finally, we define a group of isolation anomaly patterns in relational DBMSs, and detect isolation anomalies by matching these patterns in the dependency graph.

To demonstrate the effectiveness of `IsoRel`, we implement it on five widely-used relational DBMSs, i.e., MySQL [8], PostgreSQL [10], MariaDB [7], CockroachDB [3], and TiDB [15]. We further evaluate `IsoRel` on all the isolation levels that these relational DBMSs support. In total, `IsoRel` has revealed 325 isolation anomalies. From these anomalies, we identify 48 unique isolation anomalies. Among the 48 isolation anomalies, 12 anomalies cause lost update, 12 anomalies cause read-write skew, and 24 anomalies cause write skew. These isolation anomalies violate the isolation levels defined by Adya [16, 17], and have been confirmed by the respective DBMS community. For these 48 isolation anomalies, existing isolation checkers [24, 27, 28, 30, 42, 46, 49, 58, 62] can only identify 2 anomalies due to their incapability of handling relational data models and complex SQL operations. This result demonstrates that `IsoRel` can effectively handle the intricacies of relational data models and complex SQL operations. Such advancements would mitigate the risks associated with isolation anomalies and enhance the reliability of relational DBMSs.

In summary, we make the following contributions in this paper.

[2]Note that the instrumentation for `DELETE` and `REPLACE` statements may potentially affect concurrent transaction execution. However, based on our analysis of real-world relational DBMSs, we do not find such cases in real-world relational DBMSs. We will further explain this in Section 3.2.

- We propose a novel black-box isolation checker for relational DBMSs, `IsoRel`. `IsoRel` can support relational data models and complex SQL operations in relational DBMSs.
- We implement `IsoRel` and evaluate it on five widely-used relational DBMSs. `IsoRel` finds 48 unique isolation anomalies among these relational DBMSs, and most of them cannot be detected by existing isolation checkers.

2 Motivation and Preliminaries

In this section, we first present a real-world isolation anomaly in MySQL, and then formally define the SQL statements, dependencies, and isolation anomalies in relational DBMSs.

2.1 Motivating Example

Figure 1 shows a real-world isolation anomaly detected in MySQL under `Repeatable Read`. This isolation anomaly causes read-write skew [21], which should not occur under `Repeatable Read`.

This isolation anomaly involves two tables $t1$ and $t2$. Table $t1$ initially contains three rows (we use $r1$ to denote the first row with value $(1, 1)$, $r2$ to denote the second row with value $(2, 1)$, and $r3$ to denote the third row with value $(2, 2)$). Table $t2$ initially contains one row (we use $r4$ to denote this row with value $(4, 4)$). Three transactions T_1, T_2 and T_3 are executed concurrently by following the execution order of $s_{11} \rightarrow s_{12} \rightarrow s_{21} \rightarrow s_{22} \rightarrow s_{23} \rightarrow s_{24} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{31} \rightarrow s_{32} \rightarrow s_{33} \rightarrow s_{34}$. First, statement s_{12} in T_1 reads the rows in table $t1$ that satisfy $c1 < 2$ (i.e., $r1$), and then statement s_{22} in T_2 updates $c2$ to 2 for rows in $t1$ that satisfy $c1 < 2$ (i.e., $r1$). Next, statement s_{23} in T_2 inserts a row $r5$ with value $(5, 5)$ into table $t2$, and then statement s_{13} in T_1 updates $c2$ to 8 for rows in $t2$ that satisfy $c1 > 2$ (i.e., $r4$ and $r5$). Finally, statement s_{32} in T_3 reads all the rows in table $t1$ (i.e., $r1$, $r2$ and $r3$), and statement s_{33} in T_3 reads all the rows in table $t2$ (i.e., $r4$ and $r5$). We can infer that T_2 read-write depends on T_1 through $r1$ (we use `rw(`$r1$`)` to denote read-write dependency through $r1$), and T_1 write-write depends on T_2 through $r5$ (we use `ww(`$r5$`)` to denote write-write dependency through $r5$). This causes a read-write and write-write dependency cycle between T_1 and T_2 in the dependency graph that indicates read-write skew. This may cause incorrect database states.

It is difficult to infer dependencies among transactions in relational DBMSs. In this example, tables $t1$ and $t2$ are constructed without a *key-value* structure, and SQL statements involve complex SQL operations, i.e., range queries. However, existing isolation checkers [24, 27, 28, 30, 42, 46, 49, 58, 62] are limited to *key-value*-like data models and the associated $read(key)$ and $write(key, value)$ operations. They cannot identify transaction dependencies in such cases. For example, they cannot identify that s_{12} retrieves $r1$ from its query result, because the query result of s_{12} is 1, and both $r1$ and $r2$ have the value of 1 for $c2$. They also cannot identify that s_{22} updates $r1$, since $t1$ is not a *key-value* structure and allows different rows to store the same value (e.g., $r1$ and $r3$ have the same value of 2 for $c2$). Isolation checkers for *key-value*-like data models cannot determine which rows a transaction has updated by identifying unique values. As a result, they cannot infer that T_2 read-write depends on T_1 through $r1$. Similarly, they cannot infer that T_1 write-write depends on T_2 through $r5$, and cannot detect this isolation anomaly.

2.2 Transactions and SQL Statements

A database in a relational DBMS contains one or more tables, and each table manages a set of rows. We denote the set of rows managed by a database in relational DBMSs as $ROW = \{r_1, r_2, ...\}$. We denote the data of all the rows in ROW as a set of row values RV.

Transactions manipulate rows in the database utilizing SQL statements. A transaction usually contains a list of SQL statements that explicitly start with a `BEGIN` statement, and end with a `COMMIT`

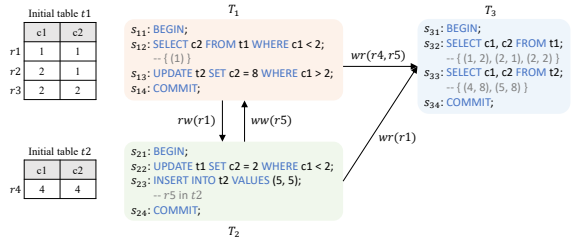


Fig. 1. Three concurrent transactions cause read-write skew under `Repeatable Read` in MySQL. The execution order for the SQL statements in three transactions is as follows: $s_{11} \rightarrow s_{12} \rightarrow s_{21} \rightarrow s_{22} \rightarrow s_{23} \rightarrow s_{24} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{31} \rightarrow s_{32} \rightarrow s_{33} \rightarrow s_{34}$.

or `ROLLBACK` statement[3]. We denote a transaction as $T_i = [s_{i1}, s_{i2}, ..., s_{im}]$, in which s_{ij} represents the jth statement of T_i ($1 \leq j \leq m$), and m is the number of statements in T_i. We denote a set of concurrent transactions as $CT = \{T_1, T_2, ..., T_n\}$, in which n is the number of concurrent transactions.

A *SQL statement* can have one of the following types, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `REPLACE`. For a `SELECT` statement, it reads one or more rows in the database. We denote the read on one row as $R(r, v)$, which represents a read operation on row r that reads data v. Thus, a `SELECT` statement is equivalent to one or more read operations. Formally, we denote a `SELECT` statement with a predicate P as $Stmt_R = \{R(r, v) | r \in ROW, v \in RV, r \ matches \ P\}$, in which predicate P represents the `WHERE` clause in the `SELECT` statement. For an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, it writes one or more rows in the database. Similarly, we denote the write on one row as $W(r, v)$, which represents a write operation on row r that writes data v. Thus, an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement is equivalent to one or more write operations. Formally, we denote an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement with a predicate P as $Stmt_W = \{W(r, v) | r \in ROW, v \in RV, r \ matches \ P\}$.

When a write statement writes a row r, it creates a new version of row r. We denote a version of row r as v_r^i. Each row r in the database has its own *write transaction set* $WT_r = \{T | W(r, v) \in s_{ij}, s_{ij} \in T, T \in CT, v \in RV\}$, which represents the set of transactions that write row r. For each row r, WT_r satisfies the strict total order relation $<$. That said, the transactions in WT_r have an execution order.

2.3 Transaction Execution Histories and Dependency Graphs

A transaction execution *history* captures the transaction execution process, which comprises the transactions that clients submit and the results that the DBMS returns. We denote the history as $H = (CT, SO)$, in which the *session order* $SO \subseteq CT \times CT$ represents the order of transactions in each session. Note that the `SELECT` statements in CT include the query results returned by the DBMS.

A *dependency graph* represents the dependency relation among transactions CT in the history H. We extend the dependency model defined by Adya [16, 17] to support relational data models and SQL statements. There are three types of *dependencies* among transactions: write-read dependency (`wr`), write-write dependency (`ww`), and read-write dependency (`rw`) [16, 17].

- **Write-read dependency (`wr`).** If a write statement $Stmt_W$ in transaction T_i writes a version of row r, and a read statement $Stmt_R$ in transaction T_j ($T_i \neq T_j$) reads the version of row r written by $Stmt_W$, then T_j write-read depends on T_i through row r. Formally, $T_i \xrightarrow{wr(r)} T_j$ if $T_i \neq T_j \wedge \exists s_{ik} \in T_i, s_{jl} \in T_j, v \in RV, W(r, v) \in s_{ik} \wedge R(r, v) \in s_{jl}$.

[3]Except for explicit transactions that start with a `BEGIN` statement and end with a `COMMIT` or `ROLLBACK` statement, a SQL statement (e.g., `UPDATE`) executed under the autocommit mode can be regarded as an implicit transaction. For the sake of clarity in our presentation, we simply treat an implicit transaction as if it were enclosed by `BEGIN` and `COMMIT` statements.

- **Write-write dependency (ww).** If a write statement $Stmt_W^i$ in transaction T_i writes a version of row r, and a write statement $Stmt_W^j$ in transaction T_j writes the next version of row r, then T_j write-write depends on T_i through row r. Formally, $T_i \xrightarrow{ww(r)} T_j$ if $T_i \neq T_j \wedge T_i \in WT_r \wedge T_j \in WT_r \wedge \nexists T_k \in WT_r, T_i < T_k < T_j$.
- **Read-write dependency (rw).** If a read statement $Stmt_R$ in transaction T_i reads a version of row r, and a write statement $Stmt_W$ in transaction T_j writes the next version of row r, then T_j read-write depends on T_i through row r. Formally, $T_i \xrightarrow{rw(r)} T_j$ if $T_i \neq T_j \wedge \exists T_k, T_k \xrightarrow{wr(r)} T_i \wedge T_k \xrightarrow{ww(r)} T_j$.

We denote a dependency graph by $G = (CT, SO, WR, WW, RW)$, in which (CT, SO) represents a history, and WR, WW and RW represent the set of `wr`, `ww` and `rw` dependencies inferred from the history (CT, SO), respectively. In the dependency graph, each vertex represents a transaction, and each edge represents a transaction dependency. If T_j `wr`, `ww` or `rw` depends on T_i, there is an edge from T_i to T_j with a label of `wr`, `ww` or `rw`. If $(T_i, T_j) \in SO$, there is an edge from T_i to T_j with a label of `so`.

2.4 Isolation Anomalies

Based on the isolation anomaly definition of Adya [16, 17], we define the isolation anomaly patterns for the relational data models, including cycle-related isolation anomaly patterns and cycle-independent isolation anomaly patterns.

Cycle-related isolation anomalies, i.e., G0, G1b, G1c, and G2-item, are defined based on the dependency graph [16, 17]. We define the isolation pattern of G0 as a cycle only including write-write dependencies in the dependency graph. We define the isolation anomaly pattern of G1b as a cycle caused by a `wr` dependency and an `rw` dependency. We define the isolation anomaly pattern of G1c as a cycle only including `ww` and `wr` dependencies. We define the isolation anomaly pattern of G2 as a cycle including `rw` dependencies in the dependency graph.

G2-item can be subdivided into many types of isolation anomalies, e.g., lost update [21], read-write skew [48], write skew [21], non-repeatable read [21], intermediate read [48], and read skew [21], according to the dependency types and the rows that form the dependencies in the dependency graph. Due to the space limitation, we only explain lost update, read-write skew, and write skew.

- **Lost update.** *lost update* occurs when the update of a statement in a transaction is lost [21]. A cycle caused by `rw` and `ww` dependencies through the same row in the dependency graph indicates a lost update. Take the following history as an example: a read statement $Stmt_R^1$ in T_1 reads row r, then a write statement $Stmt_W^2$ in T_2 updates row r, and finally a write statement $Stmt_W^1$ in T_1 updates row r based on the read of $Stmt_R^1$ and commits. The update of $Stmt_W^2$ is lost.
- **Read-write skew.** *Read-write skew* occurs when there is a cycle caused by `rw` and `ww` dependencies through the different rows in the dependency graph [48]. For example, a read statement $Stmt_R^1$ in T_1 reads row r_1. Then a write statement $Stmt_W^2$ in T_2 writes row r_1 and r_2, and commits. Finally, a write statement $Stmt_W^1$ in T_1 writes row r_2. The dependency graph among these transactions includes a cycle including `rw` dependency through row r_1 and `ww` dependency through row r_2. The motivating example in Figure 1 is read-write skew.
- **Write skew.** *Write skew* occurs when row r_1 and r_2 have a constraint, and the constraint is violated [21]. A cycle caused by two `rw` dependencies in the dependency graph indicates a write skew. For example, a read statement $Stmt_R^1$ in T_1 reads row r_1. Then a read statement $Stmt_R^2$ in T_2 reads row r_2, and a write statement $Stmt_W^2$ in T_2 writes row r_1 and commits. Finally, a write statement $Stmt_W^1$ in T_1 writes row r_2. The constraint between row r_1 and r_2 is violated since $Stmt_W^1$ updates row r_2 based on the old version of row r_1 that $Stmt_R^1$ reads.

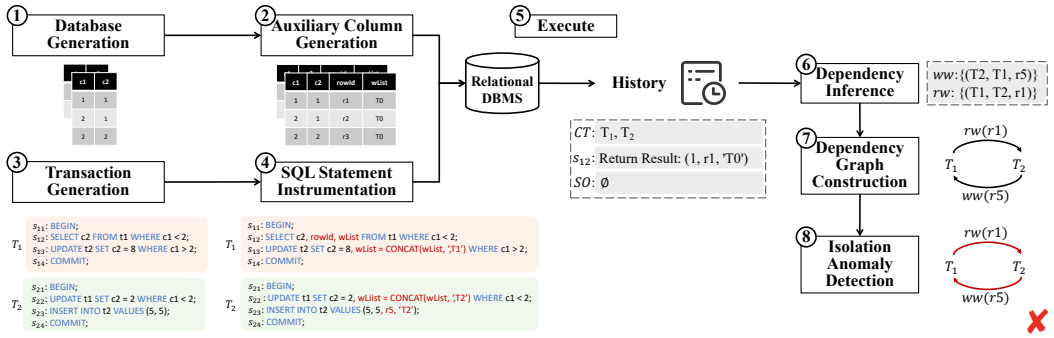


Fig. 2. The architecture of IsoRel.

Cycle-independent isolation anomaly, i.e., G1a, is defined based on the execution history of transactions [16, 17]. G1a occurs when a read statement in T_2 reads the version of row r written by a write statement in T_1 but T_1 is aborted. We define the isolation anomaly pattern of G1a as the wr dependency between transaction T_1 and T_2 in the dependency graph, where T_1 writes a row r and T_2 read the same row r, T_1 rollbacks, and T_2 commits.

3 Isolation Checker for Relational DBMSs

To detect isolation anomalies in relational DBMSs, we propose IsoRel, a novel black-box isolation checker for relational DBMSs. IsoRel generates transactions involving relational data models and complex SQL operations, infers dependencies among the transactions by adding auxiliary columns into each table and instrumenting SQL statements in an isolation-agnostic manner, and detects isolation anomalies based on isolation anomaly patterns for relational DBMSs. If an isolation anomaly is detected, IsoRel generates an anomaly report for subsequent analysis.

Figure 2 shows the architecture of IsoRel. First, we generate an initial database involving relational data models (①). Then, we add an auxiliary column *rowId* into each table to uniquely identify each row, and add an auxiliary column *wList* into each table to maintain a list of transactions that create or modify a row (②). Based on the table structures, we generate transactions involving complex SQL operations (③). Then, we instrument each read operation to additionally retrieve *rowId*s and *wList*s of their retrieved rows, and instrument each write operation to update *wList*s for their modified rows (④). We randomly select an isolation level and submit the transactions concurrently to the target relational DBMS for execution (⑤). We record the executed transactions, which include the query results of the SELECT statements within these transactions, and the session order as the execution history of transactions. Note that the query results include *rowId*s and *wList*s of the accessed rows in each SQL statement. Based on the history, we infer the dependencies among these transactions (⑥), and construct the dependency graph (⑦). Finally, we detect isolation anomalies by matching our predefined isolation anomaly patterns in the dependency graph. If IsoRel finds an isolation anomaly, it generates an anomaly report (⑧).

In the following sections, we first explain database generation and auxiliary column generation (Section 3.1), and then explain transaction generation and SQL statement instrumentation (Section 3.2). Finally, we explain dependency inference (Section 3.3), and isolation anomaly detection (Section 3.4).

3.1 Database Generation

We generate a database in two steps. First, we generate an initial database that contains one or more tables with some data. Second, we add auxiliary columns into each table.

3.1.1 Initial Database Generation. Currently, a lot of approaches have been proposed to generate databases [23, 25, 38, 40, 45]. Since database generation is not the focus of our work, we generate databases based on existing approaches, and only briefly describe it for completeness.

We generate a database involving relational data models by randomly creating tables and inserting some random data. We utilize `CREATE TABLE` statements to generate at most *maxTable* tables and at most *maxColumn* columns for each table. We randomly select a data type, e.g., INTEGER and TEXT, add column constraints, e.g., PRIMARY KEY, UNIQUE, NOT NULL, and CHECK, and add column attributes, e.g., DEFAULT and AUTOINCREMENT, for each column. Then, we insert at most *maxRow* rows into each table by utilizing `INSERT` statements. The row data are generated randomly. For example, in Figure 2, we generate an initial database with two tables. Each table has two columns. Since a recent study has found that most of the transaction bugs can be triggered on simple databases [33], we set *maxTable*, *maxColumn* and *maxRow* as 3, 3 and 5 by default.

3.1.2 Auxiliary Column Generation. To infer dependencies among transactions in relational DBMSs, we add two auxiliary columns into each table. Specifically, we add an auxiliary column *rowId* into each table to uniquely identify each row, and add an auxiliary column *wList* into each table to maintain a list of transactions that create or modify a row.

Unique rowId. Relational DBMSs lack a default *key-value* structure within their tables. This makes it difficult to infer dependencies among transactions, since we cannot identify which rows are modified by a write operation (i.e., `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statement) and which rows are retrieved by a read operation (i.e., `SELECT` statement). To address this problem, we add an auxiliary column *rowId* with data type `TEXT` into each table by using an `ALTER TABLE` statement, and assign a unique ID to each row in the tables, which is represented as "r" followed by a unique number[4]. We manually maintain *rowId* columns to ensure that *rowId* is unique for rows in all the tables. Thus, we can uniquely identify each row in the tables through *rowId*.

Write transaction list. In relational DBMSs, we cannot directly identify which rows are modified by a transaction, which complicates dependency inference. To address this problem, we add an auxiliary column *wList* into each table by using the `ALTER TABLE` statement to maintain a list of transactions that create or modify a row.

Specifically, we first identify the generated transactions by assigning a unique ID $T_{id}(id \geq 1)$ to each transaction. Then we add an auxiliary column *wList* with data type `TEXT` and initial value "`T0`" for all the initial rows in each table. When a transaction creates or modifies a certain row r, i.e., executing `INSERT`, `UPDATE`, or `REPLACE` statement on row r, its T_{id} is appended to *wList* of row r. For example, if transaction T_1 updates the initial data of a certain row, then the *wList* of this row will change from "`T0`" to "`T0,T1`". If a transaction T_2 updates a certain row with *wList* "`T0,T1`", then the *wList* of this row will change to "`T0,T1,T2`". Thus, the *wList* of a row uniquely identifies the transactions that made changes to this row.

To generate unique number in *rowId* for each row in a table, we use `UUID()` in MySQL, MariaDB, and TiDB, and `gen_random_uuid()` in PostgreSQL and CockroachDB. To maintain an append-only transaction ID list in *wList*, we use `CONCAT()` function in MySQL, MariaDB, CockroachDB, and TiDB, and "`||`" operator in PostgreSQL.

Note that our auxiliary columns are isolation-agnostic. We manually maintain the uniqueness of *rowId*s in these auxiliary columns, and do not add any column constraints to them. Thus, the added auxiliary columns will not affect the concurrent execution of the involved transactions.

[4]We use *rowId* to uniquely identify each row in the tables. Therefore, we only require that *rowId* is unique for each row. We can also use other data types, e.g., `INTEGER`, to represent *rowId*. Here, we use the "r" prefix only for better readability.

3.2 Transaction Generation and SQL Statement Instrumentation

Existing approaches have widely explored SQL statement generation [1, 13, 26, 54, 66]. We leverage these approaches to generate SQL statements, and extend them to generate transactions. We first describe transaction generation, and then describe SQL statement instrumentation in detail.

3.2.1 Transaction Generation. We generate a transaction containing a `BEGIN` statement at the beginning, at most *maxStatement* SQL statements, and a `COMMIT` or `ROLLBACK` statement at the end. The *maxStatement* is configurable. In our experiments, we set *maxStatement* to 10, according to an empirical study on transaction bugs [33]. We generate six types of SQL statements, i.e., `SELECT`, `SELECT FOR UPDATE`, `INSERT`, `UPDATE`, `DELETE`, and `REPLACE`. These SQL statements are all generated randomly. Note that for the `REPLACE` statement, only MySQL, MariaDB, and TiDB in our target relational DBMSs support it. We only implement it on these relational DBMSs.

For transaction execution, we first randomly select an isolation level. Then, we open several sessions in the target relational DBMS, set their isolation level as the selected one, and submit each transaction to a random session. Transactions are executed sequentially on a session, i.e., we only submit the next transaction for execution after the current transaction has been completed. Transactions on different sessions are executed concurrently. We record the order of transactions in each session as session order in the execution history of transactions.

3.2.2 SQL Statement Instrumentation. We use different instrumentation strategies for different types of SQL statements. Table 1 illustrates how we instrument `SELECT`, `SELECT FOR UPDATE`, `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements.

- For a `SELECT` statement, we additionally retrieve *rowId*s and *wList*s of its retrieved rows (①). For cross-table queries, we additionally retrieve *rowId*s and *wList*s of each table of its retrieved rows, e.g., $t1.rowId, t1.wList, t2.rowId, t2.wList$ (②). For subqueries and compound queries, we also instrument each sub-`SELECT` statement in the `SELECT` statement for retrieving relevant *rowId*s and *wList*s(③ and ④).
- For a `SELECT FOR UPDATE` statement, we use an instrumentation method similar to that of the `SELECT` statement to obtain the *rowId*s and *wList*s of the rows retrieved by the statement (⑤).
- For an `INSERT` statement, we add the assignments of *rowId* and *wList* where *rowId* is a unique ID for this row, and *wList* is the ID of the current transaction (⑥).
- For an `UPDATE` statement, we additionally update *wList* for its modified rows, which appends the ID of the current transaction to *wList* (⑦).
- For a `DELETE` statement, we first add a `SELECT FOR UPDATE` statement that shares the same predicate as the `DELETE` statement to retrieve the *rowId*s and *wList*s of the rows to be deleted. Then, we execute the `DELETE` statement to delete these rows (⑧). By using a `SELECT FOR UPDATE` statement, we ensure that no other transactions can interfere between the `SELECT FOR UPDATE` and the `DELETE` statements, thereby avoiding the retrieval of inconsistent row information.
- When a `REPLACE` statement is executed, the DBMS checks for an existing record with the same primary key or unique index. If found, the existing record is deleted, and the new record specified in the `REPLACE` statement is then inserted. Therefore, for a `REPLACE` statement (⑨ and ⑩), we first determine whether there is a key constraint in the target table. If the target table contains a key constraint, we additionally add a `SELECT FOR UPDATE` statement before the `REPLACE` statement, similar to the handling for a `DELETE` statement, to retrieve the *rowId*s and *wList*s of the rows that may be deleted. Then, for the `REPLACE` statement, we add the assignments of *rowId* and *wList* to the statement, like the instrumentation used for the `INSERT` statement.

Our SQL statement instrumentation is isolation-agnostic. For `SELECT`, `SELECT FOR UPDATE`, `INSERT` and `UPDATE` statements, we only add fragments to retrieve and update the auxiliary columns.

Table 1. SQL statement instrumentation for different types of SQL statements.

No.	SQL Type	SQL Example	Instrumented SQL Statement
①	SELECT	SELECT c1, c2 FROM t WHERE c2>1	SELECT c1, c2, rowId, wList FROM t WHERE c2>1
②	SELECT cross-table query	SELECT t1.c1, t2.c2 FROM t1 JOIN t2 WHERE t2.c2>1	SELECT t1.c1, t2.c2, t1.rowId, t1.wList, t2.rowId, t2.wList FROM t1 JOIN t2 WHERE t2.c2>1
③	SELECT compound query	SELECT c1, c2 FROM t1 WHERE c2>1 UNION SELECT c1, c2 FROM t2 WHERE c2>1	SELECT c1, c2, rowId, wList FROM t1 WHERE c2>1 UNION SELECT c1, c2, rowId, wList FROM t2 WHERE c2>1
④	SELECT subquery	SELECT c1, c2 FROM (SELECT c1, c2 FROM t1 WHERE c2<5) AS t2 WHERE c2>1	SELECT c1, c2, rowId, wList FROM (SELECT c1, c2, rowId, wList FROM t1 WHERE c2<5) AS t2 WHERE c2>1
⑤	SELECT FOR UPDATE	SELECT c1, c2 FROM t WHERE c2>1 FOR UPDATE	SELECT c1, c2, rowId, wList FROM t WHERE c2>1 FOR UPDATE
⑥	INSERT	INSERT INTO t(c1, c2) VALUES (1, 1)	INSERT INTO t(c1, c2, rowId, wList) VALUES (1, 1, rowId, 'TId')
⑦	UPDATE	UPDATE t SET c1=1 WHERE c2>1	UPDATE t SET c1=1, wList=CONCAT(wList, ',TId') WHERE c2>1
⑧	DELETE	DELETE FROM t WHERE c2>1	SELECT rowId, wList FROM t WHERE c2>1 FOR UPDATE DELETE FROM t WHERE c2>1
⑨	REPLACE with Key	REPLACE INTO t(c1, c2) VALUES (1, 1) /** c1 is the primary key of table t **/	SELECT rowId, wList FROM t WHERE c1=1 FOR UPDATE REPLACE INTO t(c1, c2, rowId, wList) VALUES (1, 1, rowId, 'TId')
⑩	REPLACE without Key	REPLACE INTO t(c1, c2) VALUES (1, 1) /** There is no primary key in table t **/	REPLACE INTO t(c1, c2, rowId, wList) VALUES (1, 1, rowId, 'TId')

Thus, the instrumentation does not affect the concurrent execution of the original transactions. For DELETE and REPLACE statements, we include a preceding SELECT FOR UPDATE statement to retrieve *rowId*s and *wList*s of the rows that may be deleted. This SELECT FOR UPDATE statement uses the same predicate as the subsequent DELETE or REPLACE statement and obtains exclusive locks on the rows that will be accessed by these operations. The additional SELECT FOR UPDATE statements may potentially affect concurrent transaction execution. However, our investigation of target DBMSs (i.e., MySQL [8], PostgreSQL [10], MariaDB [7], CockroachDB [3], and TiDB [15]) shows that this does not occur in our target DBMSs.

3.3 Dependency Inference

Given an execution history of a group of concurrent transactions, we infer the dependencies among these transactions by using Algorithm 1. This algorithm takes the transaction execution history recorded by our SQL statement instrumentation as input, which includes the concurrent transactions (Line 1), and the session order among the transactions executed in the same session (Line 2). We first infer wr and ww dependencies based on *rowId* and *wList* of each row in each query result (Line 5-19). Then we infer rw dependencies based on *wList*s of the same row in the query results (Line 20-34). The output of Algorithm 1 is the inferred dependencies among the transactions in the execution history, including so, wr, ww, and rw dependencies among these transactions.

We infer wr, ww and rw dependencies based on *rowId*s and *wList*s of the rows accessed in each SQL statement. The dependencies produced by SELECT, INSERT and UPDATE statements can be directly inferred. However, inferring dependencies produced by DELETE and REPLACE statements is challenging. For a DELETE statement, we cannot update and retrieve *wList*s for its modified rows after its execution, unlike INSERT and UPDATE statements, since the rows have been deleted. REPLACE statements have the same problem. A REPLACE statement may delete a corresponding row, if the related table contains a column with key constraint, and the inserted value of the key column already exists in the table. If the table does not contain a column with key constraint, or the inserted value of the key column does not exist in the table, the dependency inference for REPLACE statements is similar to that of INSERT statements.

3.3.1 Dependencies Produced by SELECT, INSERT *and* UPDATE *Statements.* SELECT, INSERT and UPDATE statements can potentially produce wr, ww and rw dependencies among transactions. We explain how to extract these dependencies as follows.

Write-read dependency. SELECT statements retrieve *rowId*s and *wList*s of their retrieved rows through our SQL statement instrumentation (Line 5-8). For a SELECT statement in transaction T_j, its query result may contain several rows. For each row, we obtain the last transaction T_i in *wList*

Algorithm 1: Dependency inference

```
   Input: history;                                          // The transaction execution history
   Output: dependencies;                                    // The dependencies among transactions
1  txes ← history.getTransacions();
2  so ← history.getSessionOrder(txes);
3  queryResults ← txes.getQueryResults();                   // The query results for all the SELECT statements
4  wr, ww, rw ← ∅;
5  foreach qr ∈ queryResults do
6      foreach row ∈ qr.getRows() do
7          rowId ← row.getRowId();
8          wList ← row.getWList();
9          readTx ← qr.getTx();
10         lastWriteTx ← wList.getLastTx();
11         if qr.getSQLStatement().associatedWithDeleteOrReplace() then
12             if readTx ≠ lastWriteTx then
13                 ww.add(< lastWriteTx, readTx, rowId >);          // lastWriteTx --ww(rowId)--> readTx
14         else if readTx ≠ lastWriteTx then
15             wr.add(< lastWriteTx, readTx, rowId >);              // lastWriteTx --wr(rowId)--> readTx
16         foreach writeTx ∈ wList do
17             preWriteTx ← wList.getPreviousTx(writeTx);
18             if preWriteTx ≠ writeTx then
19                 ww.add(< preWriteTx, writeTx, rowId >);          // preWriteTx --ww(rowId)--> writeTx
20 foreach qr_i, qr_j ∈ queryResults do
21     if !qr_i.getSQLStatement().associatedWithDeleteOrReplace() ∧ qr_i.getSQLStatement() ≠
         qr_j.getSQLStatement() ∧ containSameRowId(qr_i, qr_j) then
22         foreach rowId ∈ getSameRowIds(qr_i, qr_j) do
23             wList_i ← qr_i.getWList(rowId);
24             wList_j ← qr_j.getWList(rowId);
25             readTx ← qr_i.getTx();
26             if wList_j.equals(wList_i) then
27                 if qr_j.getSQLStatement().associatedWithDeleteOrReplace() then
28                     writeTx ← qr_j.getTx();
29                     if readTx ≠ writeTx then
30                         rw.add(< readTx, writeTx, rowId >);      // readTx --rw(rowId)--> writeTx
31             else if wList_j.startsWith(wList_i) then
32                 writeTx ← wList_j.getNextTxAfter(wList_i);
33                 if readTx ≠ writeTx then
34                     rw.add(< readTx, writeTx, rowId >);          // readTx --rw(rowId)--> writeTx
35 return so, wr, ww, rw;
```

(Line 10). We can infer that the row read by T_j is written by T_i, since T_i is the last transaction that writes this row before T_j reads it. Then, we record that T_j `wr` depends on T_i (Line 14-15).

Write-write dependency. We utilize the auxiliary column *wList* to maintain a list of transactions that create or modify a row. Thus, we directly infer `ww` dependencies based on the order of the transactions in *wLists*. Specifically, if transaction T_i immediately precedes transaction T_j in *wList* of a certain row, then T_j `ww` depends on T_i (Line 16-19).

Read-write dependency. Similar to `ww` dependencies, we can utilize the query results of `SELECT` statements to infer `rw` dependencies (Line 20-25 and Line 31-34). Specifically, for each row r that is read by two `SELECT` statements, we compare row r's *wLists* in these two `SELECT` statements (Line 20-24). If the $wList_i$ of row r read by a `SELECT` statement $Stmt_R^i$ in transaction T_i is the prefix of

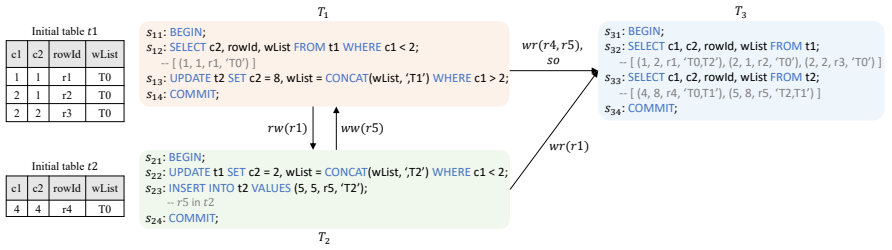


Fig. 3. The execution history of concurrent transactions in Figure 1. We can infer wr, ww and rw dependencies of SELECT, INSERT and UPDATE statements.

another $wList_j$ of row r read by a SELECT statement $Stmt_R^j$ in transaction T_j (Line 31), we can infer that the remaining (except $wList_i$) list of transactions in $wList_j$ further modify row r after executing $Stmt_R^i$. We use T_k to denote the first transaction in $wList_j$ that modifies row r after executing $Stmt_R^i$ (Line 32). We can infer that after $Stmt_R^i$ reads row r, the first transaction that writes row r is T_k. Then, T_k rw depends on T_i (Line 33-34).

Figure 3 shows the execution history of concurrent transactions in Figure 1 with our SQL statement instrumentation. T_1 and T_3 are in the same session. From the query result of s_{32}, we can infer that s_{32} reads row $r1$ written by T_2. Thus, T_3 wr depends on T_2. From the query result of s_{33}, we can infer that T_2 first writes row $r5$ and then T_1 writes $r5$. Thus, T_1 ww depends on T_2. For row $r1$, the $wList$ retrieved by s_{12} is "T0", and the $wList$ retrieved by s_{32} is "T0,T2". We can infer that for $r1$, T_1 first reads $r1$, and then T_2 writes it. Thus, T_2 rw depends on T_1.

Note that our dependency inference relies on $rowId$s and $wList$s obtained from the query results of SELECT statements within transactions. These ww and rw dependencies can be inferred only if there is a read operation on the corresponding rows after write operations. Although transactions are executed continuously, it may not always contain read operations after write operations. Thus, some dependencies may be overlooked, potentially resulting in false negatives. To mitigate this issue, we can frequently execute transactions that retrieve all data in our target relational DBMSs.

3.3.2 Dependencies Produced by DELETE Statements. DELETE statements can potentially produce ww and rw dependencies among transactions. We explain how to extract these dependencies as follows.

Write-write dependency. We retrieve the $rowId$s and $wList$s of the rows to be deleted by adding a SELECT FOR UPDATE statement before each DELETE statement. Thus, we can know the last transaction that writes a specific row before it is deleted, and infer ww dependencies. Specifically, for each row to be deleted, if the last transaction in $wList$ is T_i, and the transaction that deletes this row is T_j, then T_j ww depends on T_i (Line 11-13).

Read-write dependency. To obtain which transactions read a certain deleted row just before it is deleted, we first utilize the added SELECT FOR UPDATE statement associated with the corresponding DELETE statement to retrieve all the deleted rows' $rowId$s and $wList$s. Then, for each deleted row, we find which transaction read the same $wList$ as that of this row. Specifically, for each row deleted by transaction T_j, if the $wList$ read by transaction T_i is the same as the $wList$ read by the added SELECT FOR UPDATE statement (Line 26-27), then T_j rw depends on T_i (Line 29-30).

Figure 4 shows an execution history of two concurrent transactions T_1 and T_2. First, T_1 updates $c1$ to 1 in table t where $c1 < 5$, and then reads the rows in t where $c1 < 5$. Then, T_2 reads the $rowId$s and $wList$s in t where $c2 < 2$, and then deletes the rows in t where $c2 < 2$. From the query result of s_{22}, we can infer that for row $r1$, T_1 first writes it, and then T_2 deletes it. Thus, T_2 ww depends on T_1. For row $r1$, the $wList$ retrieved by s_{13} is "T0,T1", which is the same as the $wList$ retrieved by s_{22}. Thus, we can infer that T_2 rw depends on T_1.

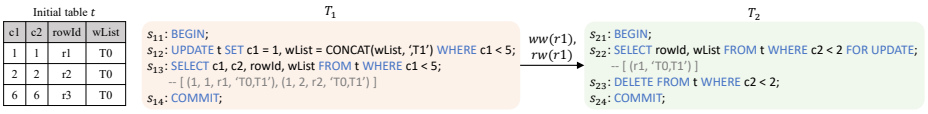


Fig. 4. We can infer ww and rw dependencies of a DELETE statement. The execution order for the SQL statements is as follows: $s_{11} \rightarrow s_{21} \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{22} \rightarrow s_{23} \rightarrow s_{24}$.

3.3.3 Dependencies Produced by REPLACE Statements. For each record to be replaced, the REPLACE statement first checks whether a row with the same primary key or unique index as the new record specified in the REPLACE statement exists in the target table. If yes, the corresponding row is deleted, and the new record specified in the REPLACE statement is inserted into the target table. If not, the new record specified in the REPLACE statement is directly inserted into the target table. Therefore, we need to infer ww and rw dependencies on the deleted rows (Line 11-13 and Line 26-30), and infer ww and wr dependencies on the inserted rows (Line 14-19).

For ww and rw dependencies on a deleted row in a REPLACE statement, we infer them in a manner similar to that used for DELETE statements. We first retrieve the columns that have key constraints in tables from the CREATE statements generated during the database generation process. Then, we retrieve the *rowId* and *wList* of the row that has the same key with the inserted record by adding a SELECT FOR UPDATE statement before the REPLACE statement. Thus, we can know the row that is deleted by the REPLACE statement. Then, we infer ww and rw dependencies in the same way as that of DELETE statements (Line 11-13 and Line 26-30). For ww and wr dependencies on the inserted rows, we infer them in the same way as that of INSERT statements (Line 14-19).

Based on the SQL statement instrumentation, we can infer ww and rw dependencies on the deleted rows between a DELETE (or REPLACE) statement and the statements that are executed before the DELETE (or REPLACE) statement. Since the deleted rows by the DELETE (or REPLACE) statement have disappeared in the database, we cannot trace the ww and wr dependencies on the deleted rows after the DELETE (or REPLACE) statement is executed anymore. However, this only leads to losing some dependencies, potentially resulting in false negatives.

3.4 Isolation Anomaly Detection

To detect isolation anomalies in relational DBMSs, we construct the dependency graph among transactions based on our inferred dependencies. Then, we detect isolation anomalies by matching isolation anomaly patterns defined in Section 2.4 in the dependency graph. If we find an isolation anomaly, we generate an anomaly report for further analysis.

In a dependency graph G, the vertices are the transactions in the execution history of transactions, and the edges are the transaction dependencies we inferred, which contain wr, ww, rw and so dependencies. If transaction T_j depends on transaction T_i (i.e., wr, ww and rw dependencies), or T_i immediately precedes T_j in the same session (i.e., so dependencies), there is a direct edge from T_i to T_j in graph G. Figure 3 shows a dependency graph of T_1, T_2 and T_3 with wr, ww, rw and so edges.

We detect isolation anomalies by matching the isolation anomaly patterns in the dependency graph. The isolation anomaly patterns define the dependency types and the rows that form the dependencies in isolation anomalies.

For cycle-related isolation anomalies, we identify strongly connected components in the dependency graph as cycles by utilizing the Tarjan algorithm [60]. For each cycle, we determine the type of isolation anomaly by matching the anomaly patterns with the dependency types of edges in the cycle. For example, in Figure 3, we detect a dependency cycle between T_1 and T_2, which contains a ww dependency and an rw dependency. Thus, we identify it as read-write skew.

Note that for SELECT FOR UPDATE statements, we do not identify some dependency cycles as isolation anomalies. Specifically, if a cycle between T_1 and T_2 includes an rw dependency produced

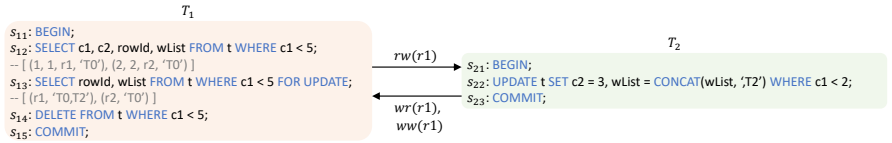


Fig. 5. We do not consider the dependency cycle as an isolation anomaly due to the `SELECT FOR UPDATE` statement. The execution order for the SQL statements is as follows: $s_{11} \rightarrow s_{12} \rightarrow s_{21} \rightarrow s_{22} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15} \rightarrow s_{23}$.

by a `SELECT` statement in T_1 and a `wr` dependency produced by a `SELECT FOR UPDATE` statement in T_1 (e.g., $T_1 \xrightarrow{rw(r)} T_2$ and $T_2 \xrightarrow{wr(r)} T_1$), it is not an isolation anomaly. This is because the `SELECT FOR UPDATE` statement is current read, instead of snapshot read, so it can read the latest data. To address this problem, if we match this pattern in the dependency graph, we do not identify it as an isolation anomaly. For example, in Figure 5, we detect a cycle between T_1 and T_2 with `rw` and `wr` dependencies, and the `wr` dependency is produced by a `SELECT FOR UPDATE` statement in T_1. Thus, we do not identify it as an isolation anomaly.

For cycle-independent isolation anomaly (i.e., G1a), we match the isolation anomaly pattern of G1a in the dependency graph to identify it. We detect the `wr` dependency between T_1 and T_2 in the dependency graph. If T_1 rollbacks and T_2 commits, we identify it as G1a.

4 Soundness Proof of `IsoRel`

A database manages a set of rows $ROW = \{r_1, r_2, ...\}$ in a relational DBMS. Each row r in the database has its own *write transaction list* $wList_r = [T_r^1, T_r^2, ...]$, in which $T_r^i \in CT$ (the set of concurrent transactions) and transaction T_r^i writes the i-th version v_r^i of row r. Specifically, when a write statement in transaction T_i creates or modifies row r, T_i is appended into $wList_r$. In such a situation, each version v_r^i of row r corresponds to a unique write transaction list $wList_r^i$. To deliver a clear presentation, we do not discuss read-write and write-write dependencies produced by `DELETE` and `REPLACE` statements in the following proof, which can be described in the similar way as Lemma 2 and Lemma 3.

Lemma 1. If a read statement $Stmt_R$ in transaction T_j retrieves a row r, and row r's corresponding write transaction list $wList_r$ is $[..., T_i]$ $(T_i \neq T_j)$ (i.e., transaction T_i is the last transaction in $wList_r$), then T_j write-read depends on T_i through row r.

Proof. Row r's corresponding write transaction list $wList_r$ is $[..., T_i]$, so there exists a write statement $Stmt_W$ in T_i that writes row r and creates this $wList_r$. Since $wList_r$ is unique for each version of row r, $Stmt_R$ must read the version written by $Stmt_W$, and the versions of row r written by other statements and transactions are not visible to $Stmt_R$ in T_j. Therefore, T_j write-read depends on T_i through row r.

Lemma 2. If a row r's write transaction list $wList_r$ is $[..., T_i, T_j, ...]$ $(T_i \neq T_j)$ (i.e., transaction T_i immediately precedes transaction T_j in $wList_r$), then T_j write-write depends on T_i through row r.

Proof. Row r's corresponding write transaction list $wList_r$ is $[..., T_i, T_j, ...]$, so there exists a write statement $Stmt_W^i$ in T_i and a write statement $Stmt_W^j$ in T_j, which update row r one after another. If a statement $Stmt_W^k$ in transaction T_k ($T_k \neq T_i$ and $T_k \neq T_j$) writes row r between $Stmt_W^i$ and $Stmt_W^j$, then $wList_r$ will become $[..., T_i, T_k, T_j, ...]$, which conflicts with the precondition (i.e., row r's write transaction list $wList_r$ is $[..., T_i, T_j, ...]$). Therefore, T_j write-write depends on T_i through row r.

Lemma 3. If a read statement $Stmt_R$ in transaction T_i retrieves a row r, and row r's corresponding write transaction list $wList_r$ is $[..., T_k]$ (i.e., transaction T_k is the last transaction in $wList_r$), and row r's $wList_r$ becomes $[..., T_k, T_j, ...]$ $(T_j \neq T_i)$ in another read statement, then T_j read-write depends on T_i through row r.

Table 2. Target relational DBMSs in our evaluation.

Relational DBMS	DB-Engines Ranking	Stars	Isolation Level	Concurrency Control
MySQL	2	9.9K	RU, RC, RR, SER	Pessimistic
PostgreSQL	4	13.9K	RC, RR, SER	Pessimistic, Optimistic
MariaDB	13	5.1K	RU, RC, RR, SER	Pessimistic
CockroachDB	67	29.8K	SER	Optimistic
TiDB	87	35.6K	RC, RR	Pessimistic, Optimistic

Proof. $Stmt_R$ in transaction T_i retrieves row r, and row r's corresponding write transaction list $wList_r$ is $[..., T_k]$, so $Stmt_R$ reads the version of row r written by T_k. Since row r's $wList_r$ becomes $[..., T_k, T_j, ...]$ in another read statement, there exists a write statement $Stmt_W$ in T_j writes the version of row r with $wList_r = [..., T_k, T_j]$. No other statements can update row r between $Stmt_R$ and $Stmt_W$. Therefore, T_j read-write depends on T_i through row r.

The above three lemmas prove that the transaction dependencies inferred by IsoRel are sound, and have no false positives. Therefore, the dependency graph $G = (CT, SO, WR, WW, RW)$, in which SO represents the session order on concurrent transactions CT, and WR, WW and RW represent the set of write-read dependencies, write-write dependencies and read-write dependencies among transactions CT, is sound and does not contain wrong dependencies. Existing works [16, 17, 30, 42, 46, 49, 58, 62] have proved that the appearance of different kinds of cycles in the dependency graph G indicates different kinds of isolation anomalies, e.g., a cycle caused by read-write and write-write dependencies through the same row in the dependency graph G indicates a lost update. In IsoRel, we adopt the Tarjan algorithm [60] to identify cycles in the dependency graph, which is sound and does not report false positives. Therefore, IsoRel is sound.

5 Evaluation

We evaluate `IsoRel` on five widely-used relational DBMSs, i.e., MySQL [8], PostgreSQL [10], MariaDB [7], CockroachDB [3], and TiDB [15], and answer the following three research questions:

- **RQ1**: How effective is `IsoRel` in detecting isolation anomalies in relational DBMSs?
- **RQ2**: What are the root causes of the isolation anomalies detected by `IsoRel`?
- **RQ3**: How does `IsoRel` compare against existing isolation checkers [42, 46, 58]?

5.1 Experimental Methodology

Target relational DBMSs. We select five widely-used relational DBMSs, i.e., MySQL [8], PostgreSQL [10], MariaDB [7], CockroachDB [3], and TiDB [15], as our target relational DBMSs, based on their DB-Engines Ranking [4] and GitHub stars [5]. Table 2 shows the detailed information. These relational DBMSs support multiple isolation levels, e.g., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable` in MySQL and MariaDB. We check all target relational DBMSs on the latest release versions when we started this evaluation, i.e., MySQL 8.2.0, PostgreSQL 16.2, MariaDB 10.11.2, CockroachDB 24.1.1, and TiDB 8.1.0.

Reproducing known isolation anomalies. We first evaluate whether `IsoRel` can reproduce known isolation anomalies revealed by existing isolation checkers [6, 42, 46]. We investigate all the relational DBMSs developers' confirmed isolation anomalies [9, 11, 14, 42] detected by existing checkers, and collect 6 known isolation anomalies. We extract the features of these anomalies from their anomaly reports, including the table structures, operation features, and dependencies. Then we run `IsoRel` on the relational DBMSs in the release versions that can trigger these isolation anomalies, and collect all the anomaly reports generated by `IsoRel`.

For each detected isolation anomaly, we manually remove and simplify the transactions and statements, which are not necessary to reproduce the anomaly. First, we remove the transactions and statements that do not affect the anomaly dependencies in the dependency graph. Second, we further remove the unnecessary tables, columns and rows, and simplify SQL statements if possible

Table 3. Known isolation anomalies rediscovered by IsoRel.

Relational DBMS	Release Version	Isolation Level	Isolation Anomaly
MySQL	v8.0.34	RR	write skew
MySQL	v8.0.34	RR	read-write skew
MySQL	v8.0.34	RR	lost update
PostgreSQL	v12.3	RR	write skew
MariaDB-Galera	v10.7.3	RR	lost update
TiDB	v2.1.7	RR	write skew

(e.g., using a simpler WHERE clause while keeping the same anomaly dependencies). On average, it takes approximately 20 minutes to manually simplify a detected anomaly. Finally, we manually compare the features of our detected isolation anomalies with those of known anomalies. If their features are the same, we conclude that IsoRel reproduces these known anomalies.

Detecting new isolation anomalies. To conduct this experiment, we randomly generate a database *db* in the target relational DBMS by using the database generation approach. For the generated database *db*, we create *numSession* concurrent sessions, which is randomly chosen between 2 and 5. For each session, we randomly generate transactions, and submit them one by one to the target DBMS. We run these sessions for 10 minutes, and record the execution history of transactions. We then run dependency inference approach and isolation anomaly detection approach to detect isolation anomalies. We repeat the above steps until we have run for a total of 6 hours for a target DBMS. In our experiment, we repeat these steps 35 times for each DBMS.

IsoRel can support all the isolation levels supported by our target DBMSs, e.g., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable` in MySQL and MariaDB. To demonstrate its effectiveness, we run IsoRel on the five relational DBMSs under all supported isolation levels. Once an isolation anomaly is detected, IsoRel produces an anomaly report with essential information for reproducing and analysis, including the type of the isolation anomaly, the constructed dependency graph, the involved transactions, the query results of the `SELECT` statements in the transactions, the initial tables, and the isolation level. Then, we manually simplify the isolation anomaly by utilizing the simplification process discussed earlier. Finally, we manually identify unique isolation anomalies. For any two anomalies, if the number, order, or types of the statements in their involved transactions are different, we consider them to be unique.

5.2 Isolation Anomaly Detection Results

Reproducing known isolation anomalies. IsoRel successfully reproduces 6 known isolation anomalies [9, 11, 14, 42]. Details about these isolation anomalies are shown in Table 3, including the relational DBMSs where they are revealed, the release versions of these relational DBMSs, the isolation levels, and the types of these anomalies. Specifically, IsoRel reproduces three known anomalies in MySQL 8.0.34 [9], one known anomaly in PostgreSQL 12.3 [11], one known anomaly in MariaDB-Galera 10.7.3 [42], and one known anomaly in TiDB 2.1.7 [14]. All these isolation anomalies occur under `Repeatable Read`. The types of these anomalies are diverse, including write skew, read-write skew and lost update.

Detecting new isolation anomalies. In total, IsoRel has revealed 325 isolation anomalies in the five target relational DBMSs, as shown in Table 4. From these anomalies, we identify 48 unique isolation anomalies, all of which are revealed under `Repeatable Read`. We have not yet observed isolation anomalies under `Read Uncommitted`, `Read Committed`, and `Serializable` isolation levels. Among the 48 isolation anomalies, 14 anomalies are found in MySQL, 6 anomalies are found in PostgreSQL, 14 anomalies are found in MariaDB, and 14 anomalies are found in TiDB. For the 48 anomalies, 12 anomalies are lost update, 12 anomalies are read-write skew, and 24 anomalies are write skew.

Table 4. Isolation anomalies detected by IsoRel.

Relational DBMS	#Txn	Isolation Level				Total (Unique)	Lost Update	Read-Write Skew	Write Skew
		RU	RC	RR	SER				
MySQL	5774	0	0	57	0	57 (14)	4	4	6
PostgreSQL	2536	-	0	8	0	8 (6)	0	0	6
MariaDB	4515	0	0	92	0	92 (14)	4	4	6
CockroachDB	4090	-	-	-	0	0 (0)	0	0	0
TiDB	5802	-	0	168	-	168 (14)	4	4	6
Total	22717	0	0	325	0	325 (48)	12	12	24

All the 48 unique anomalies violate the `Repeatable Read` isolation level defined by Adya [16, 17]. We have further investigated these anomalies by inspecting the corresponding DBMS documents and discussing with DBMS developers, and confirmed that all these anomalies are true positives and indeed occur in these DBMSs. However, they are allowed to occur in these target DBMSs due to certain design choices for now.

5.3 Analyzing Isolation Anomalies in Relational DBMSs

We further conduct a study on the 48 unique isolation anomalies detected by `IsoRel` to thoroughly analyze their root causes. Each anomaly is independently studied and classified by three authors to mitigate subjectivity. The analysis results are discussed at length until a consensus is reached on the root causes of the anomalies. From the 48 unique isolation anomalies, we find that they are all caused by no lock conflicts. We further divide into two root causes as follows.

No lock conflicts due to the timing of `COMMIT`. 36 (75%) isolation anomalies are caused by no lock conflicts due to the timing of `COMMIT` statements. Among the 36 isolation anomalies, 12 anomalies are lost update, 12 anomalies are read-write skew, and 12 anomalies are write skew.

For lost update and read-write skew, the cycles that define them contain at least a `ww` dependency in the dependency graph. The `ww` dependency is produced by two transactions writing the same row one after another. Because the previous transaction commits before the latter transaction writes the row, the lock on the modified row is released, causing no lock conflicts. Thus, although these anomalies satisfy the anomaly definition of Adya [16, 17], they are allowed to occur in our target relational DBMSs under `Repeatable Read`. Conversely, if the previous transaction does not commit, and the latter transaction writes the row, then a lock conflict occurs. Such isolation anomalies are proscribed to occur in our target relational DBMSs under `Repeatable Read`.

For write skew, two transactions write different rows in the same table, and the relational DBMSs apply gap locks on these rows by `UPDATE` and `DELETE` statements. Similarly, since the previous transaction commits before the latter transaction writes the rows, lock conflict does not occur. These anomalies are allowed to occur in our target relational DBMSs under `Repeatable Read`.

Figure 1 shows a read-write skew detected in MySQL under `Repeatable Read`. In this case, T_2 `rw` depends on T_1 through row $r1$, and T_1 `ww` depends on T_2 though row $r5$. Thus, this anomaly satisfies read-write skew defined by Li [48], which violates the `Repeatable Read` isolation level defined by Adya [16, 17]. However, it is allowed to occur in MySQL since T_2 first inserts row $r5$ into table $t2$, and after T_2 commits, T_1 updates $r5$ in table $t2$.

No lock conflicts due to accessing different rows. 12 (25%) isolation anomalies are caused by no lock conflicts due to accessing different rows. These anomalies are all write skew, which is defined as a cycle caused by two `rw` dependencies through different rows in the dependency graph. If lock conflicts do not occur between the transactions that produce the two `rw` dependencies, these anomalies are allowed to occur in our target relational DBMSs under `Repeatable Read`. We further categorize the root cause into two types and describe them as follows.

- **Access different rows in different tables.** In this case, two transactions T_1 and T_2 write different rows in different tables, so lock conflicts do not occur between T_1 and T_2.

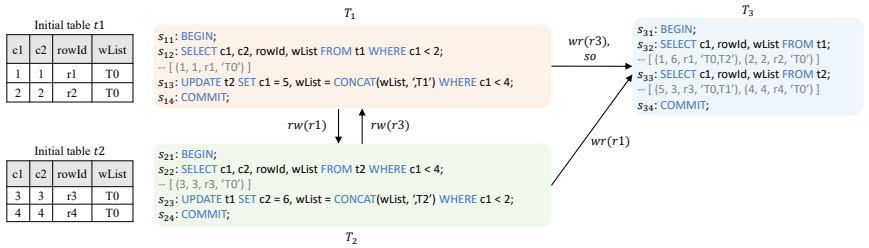


Fig. 6. The write skew in MySQL that accesses different rows in different tables. The execution order for the SQL statements is as follows: $s_{11} \rightarrow s_{21} \rightarrow s_{12} \rightarrow s_{22} \rightarrow s_{13} \rightarrow s_{23} \rightarrow s_{14} \rightarrow s_{24} \rightarrow s_{31} \rightarrow s_{32} \rightarrow s_{33} \rightarrow s_{34}$.

- **Access different rows in the same table without gap locks.** In this case, two transactions T_1 and T_2 write different rows in the same table. If the relational DBMS does not apply gap locks (e.g., TiDB does not apply gap locks), lock conflicts do not occur between T_1 and T_2.

Figure 6 shows a write skew detected in MySQL under `Repeatable Read`. In this case, T_1 and T_3 are in the same session. T_2 `rw` depends on T_1 through row $r1$, and T_1 `rw` depends on T_2 through row $r3$. Thus, this isolation anomaly satisfies write skew defined by Berenson [21]. However, it is allowed to occur in MySQL, since row $r3$ and $r1$ are in different tables, and no lock conflict occurs between T_1 and T_2.

5.4 Comparison with Existing Isolation Checkers

Existing isolation checkers [24, 27, 28, 30, 42, 46, 49, 58, 62] mainly work on *key-value*-like data models, in which data is organized in a *key-value* format. These checkers are limited to detecting isolation anomalies only within simple *key-value*-like data models and the associated $read(key)$ and $write(key, value)$ operations. They cannot infer `wr`, `ww` and `rw` dependencies among transactions that involve complex SQL operations and relational data models in relational DBMSs, which IsoRel can infer. Therefore, we cannot directly compare IsoRel with existing checkers when these checkers are applied to relational DBMSs. Thus, we analyze whether three existing checkers (Elle [46], Cobra [58], PolySI [42]) can conceptually detect the 48 isolation anomalies revealed by `IsoRel`.

For an isolation anomaly, we conclude that existing checkers cannot detect it from the following two conditions. First, the anomaly involves table structure different from *key-value* structure. For example, in Figure 6, the test case contains two tables, which cannot be triggered by *key-value* data model. Second, the anomaly involves complex SQL operations different from $read(key)$ and $write(key, value)$ operations. For example, in Figure 3, the SQL statements s_{12}, s_{13}, and s_{22} contain range query, which cannot be supported by $read(key)$ and $write(key, value)$ operations. We find that for the 48 isolation anomalies, Elle and PolySI can only detect 2 anomalies, and Cobra cannot detect all the 48 anomalies. For the 46 isolation anomalies that Elle and PolySI cannot detect, 4 anomalies involve table structure different from *key-value* structure, and the rest 42 anomalies involve complex SQL operations different from $read(key)$ and $write(key, value)$ operations.

6 Discussion

In this section, we discuss the limitations of our approach and the potential future work.

6.1 Limitations

Although `IsoRel` can effectively detect isolation anomalies in relational DBMSs, it cannot support some SQL operations and some transaction dependencies, potentially resulting in false negatives.

Unsupported SQL operations. Although `IsoRel` supports many SQL operations (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE` and `REPLACE`), and complex SQL features (e.g., subqueries, compound queries and cross-table queries), it does not support aggregate functions and views, since the rows

accessed by aggregate functions are not contained in the query results. As a result, `IsoRel` cannot retrieve these accessed rows from the query results, and fails to infer the associated dependencies.

Overlooking some dependencies. `IsoRel` can overlook some dependencies, resulting in false negatives. First, `IsoRel` cannot infer `ww` and `wr` dependencies on the deleted rows after a `DELETE` (or `REPLACE`) statement is executed, since the deleted rows have disappeared in the database. Second, `IsoRel` cannot infer all dependencies of compound queries and subqueries since some rows may be accessed but are not in the query results. Third, the instrumented information of write statements in the rolled-back transactions will be deleted by DBMSs along with the rolled-back transactions, and `IsoRel` cannot obtain these information anymore after rollbacks. Thus, `IsoRel` can overlook some dependencies associated with these rolled-back transactions. Fourth, `IsoRel` cannot infer predicate dependencies. To infer predicate dependencies, we need to know whether a write statement changes the matches of a predicate read. However, we have not yet identified a suitable approach to do so in an isolation-agnostic manner, preventing `IsoRel` from currently supporting predicate dependencies.

6.2 Future Work

Automatic isolation anomaly simplification. We manually simplify the detected isolation anomalies, which requires considerable effort. The manual simplification process can potentially be automated. One potential challenge for the automated simplification process is how to confirm the simplified transactions can still trigger the same isolation anomaly due to the nondeterminism of transaction execution.

Speeding up isolation anomaly detection. Different relational DBMSs have some commonalities of isolation anomalies in them. It should be possible to create some templates of the isolation anomalies detected in one relational DBMS. These templates can potentially lead to anomalies in other relational DBMSs. Thus, we can use these templates to speed up isolation anomaly detection in other relational DBMSs, and help converge more quickly.

Extending `IsoRel` by adopting existing anomaly detection algorithms. In our work, building the dependency graph from the transaction execution history in relational DBMSs is our main contribution. We further adopt a simple algorithm (i.e., graph traversal by utilizing the Tarjan algorithm [60]) to detect cycles in the dependency graph. Existing isolation anomaly detection algorithms [24, 27, 28, 30, 42, 46, 49, 58, 62] also aim to detect cycles in the dependency graph, among which there are some efficient anomaly detection algorithms, e.g., resolving uncertain dependencies through SMT solvers [42, 58, 62]. Therefore, once we build the dependency graph in relational DBMSs through our SQL statement instrumentation and dependency inference approach, we can potentially adopt existing anomaly detection algorithms (e.g., Cobra [58] and PolySI [42]) to detect isolation anomalies in relational DBMSs to improve detection efficiency.

7 Related Work

Isolation level specification. ANSI [2] defines four isolation levels, i.e., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable`, and the corresponding four isolation anomalies that are proscribed at each isolation level, i.e., dirty write, dirty read, non-repeatable read, and phantom. Berenson et al. [21] show that the isolation levels defined by ANSI are incomplete. They refine the definition of isolation levels, and propose `Cursor Stability`, `Snapshot Isolation`, and the formal specification of isolation anomalies. Adya et al. [16, 17] define the dependencies among transactions and propose weaker isolation levels based on transaction dependency.

Isolation checking. Some isolation checkers have been proposed to check the isolation levels utilizing *key-value*-like data models. Elle [46] infers transaction dependencies based on the incrementable list, and detects isolation anomalies based on dependency graph in *key-value*-like data

models. Cobra [58] utilizes polygraph, which utilizes constraints to capture unknown dependencies, and uses SMT solvers to detect cycles. PolySI [42] proposes a generalized polygraph and the formal definition of `Snapshot Isolation`. It accelerates the checking of `Snapshot Isolation` in *key-value*-like data models through SMT solvers. Viper [62] proposes a BC-polygraph, and accelerates the checking of `Snapshot Isolation` in *key-value*-like data models through SMT solvers. Emme [30] is a white-box checker that retrieves transaction timestamps and recovers a version certificate to check isolation levels in DBMSs that use multi-version concurrency control timestamp ordering protocol to guarantee serializability. However, these isolation checkers focus on *key-value*-like data models, and can not be applied to relational data models.

Transaction testing. To improve the quality of transaction implementation, some transaction testing approaches have been proposed. DT^2 [32] executes the same group of transactions in multiple DBMSs and compares their execution results to detect transaction bugs. Troc [35] builds a transaction test oracle to determine whether a group of transactions executes correctly. TxCheck [44] constructs semantically equivalent test cases and compares their execution results to detect transaction bugs. WriteCheck [34] proposes write-specific serializability, and uses this property to detect write-specific serializability violations. However, these works cannot detect isolation anomalies under non-deterministic executions in relational DBMSs.

DBMS testing. Many approaches have been proposed to test DBMSs. SQLsmith [13] detects crash bugs by executing randomly generated SQL statements. Rigger et al. propose SQLancer [12] and several approaches, e.g., PQS [53], TLP [52] and NoREC [51], to detect logic bugs in `SELECT` statements. QPG [18] utilizes query plans to guide DBMS testing. DQE [55] executes `SELECT`, `UPDATE` and `DELETE` statements with the same predicate to detect logic bugs. Radar [56] differentially tests the original database and raw database that wipes out the metadata to detect metadata-related logic bugs. DDLCheck [57] examines the behavior differences on the same database schema created by different DDL sequences to detect schema-related logic bugs. Grand [65], GDsmith [41] and RD^2 [61] adopt differential testing to detect logic bugs in graph DBMSs. Qudi [63], GQT [43] and GRev [50] further utilize equivalent queries [59] to detect logic bugs in graph DBMSs. DOT [64] further detects optimization bugs in graph DBMSs. However, these works cannot detect isolation anomalies in relational DBMSs.

8 Conclusion

Incorrect implementations of transaction mechanisms can cause isolation anomalies and undermine the isolation levels claimed by relational DBMSs. In this paper, we propose `IsoRel`, a black-box isolation checker for relational DBMSs, which can support relational data models and complex SQL operations. The core idea of `IsoRel` is to instrument SQL statements in an isolation-agnostic manner and extract the accessed rows of each SQL statement to track dependencies among transactions in relational DBMSs. We evaluate `IsoRel` on five widely-used relational DBMSs and have found 48 isolation anomalies, most of which cannot be detected by existing isolation checkers.

9 Data Availability

`IsoRel` has been made available at https://figshare.com/s/c79c57db38052eab488b.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62072444, 62302493), Major Project of ISCAS (ISCAS-ZD-202302), Basic Research Project of ISCAS (ISCAS-JCZD-202403), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044). This work was also partially supported by Huawei.

References

[1] 2023. go-randgen. https://github.com/pingcap/go-randgen.
[2] 2024. The ANSI Isolation Levels. http://www.adp-gmbh.ch/ora/misc/isolation_level.html.
[3] 2024. CockroachDB. https://www.cockroachlabs.com.
[4] 2024. DB-Engines. https://db-engines.com/en/ranking.
[5] 2024. GitHub. https://github.com.
[6] 2024. Jepsen. https://github.com/jepsen-io/jepsen.
[7] 2024. MariaDB. https://mariadb.org.
[8] 2024. MySQL. https://www.mysql.com.
[9] 2024. MySQL 8.0.34. https://jepsen.io/analyses/mysql-8.0.34?file=mysql-8.0.34.
[10] 2024. PostgreSQL. https://www.postgresql.org.
[11] 2024. PostgreSQL 12.3. https://jepsen.io/analyses/postgresql-12.3?file=postgresql-12.3.
[12] 2024. SQLancer. https://www.manuelrigger.at/dbms-bugs.
[13] 2024. SQLsmith. https://jepsen.io.
[14] 2024. TiDB 2.1.7. https://jepsen.io/analyses/tidb-2.1.7?file=tidb-2.1.7.
[15] 2024. TiDB, PingCAP. https://pingcap.com.
[16] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph. D. Dissertation. Massachusetts Institute of Technology.
[17] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of International Conference on Data Engineering (ICDE)*. 67–78.
[18] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2060–2071.
[19] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proceedings of the VLDB Endowment (VLDB)* 7, 3 (2013), 181–192.
[20] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 15:1–15:45.
[21] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1–10.
[22] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc.
[23] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 341–352.
[24] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 165:1–165:28.
[25] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1097–1107.
[26] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 18, 12 (2006), 1721–1725.
[27] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 458–472.
[28] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*. 58–71.
[29] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control (SIGFIDET)*. 249–264.
[30] Jack Clark, Alastair F Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 754–768.
[31] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.
[32] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 35:1–35:12.
[33] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 163:1–163:13.

[34] Ziyu Cui, Wensheng Dou, Yu Gao, Rui Yang, Yingying Zheng, Jiansen Song, Yuan Feng, and Jun Wei. 2025. Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems. *Proceedings of the VLDB Endowment (VLDB)* (2025).

[35] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 1123–1135.

[36] Jim Gray. 1981. The Transaction Concept: Virtues and Limitations. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 144–154.

[37] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Elsevier.

[38] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 243–252.

[39] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)* 15, 4 (1983), 287–317.

[40] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1243–1246.

[41] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 163–174.

[42] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-box Checking of Snapshot Isolation in Databases. *Proceedings of the VLDB Endowment (VLDB)* 16, 6 (2023), 1264–1276.

[43] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. Article 46, 12 pages.

[44] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 397–417.

[45] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–247.

[46] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. In *Proceedings of the VLDB Endowment (VLDB)*, Vol. 14. 268–280.

[47] Philip M Lewis, Arthur Bernstein, and Michael Kifer. 2002. Databases and Transaction Processing: An Application-Oriented Approach. *ACM SIGMOD Record* 31, 1 (2002), 74–75.

[48] Haixiang Li, Xiaoyan Li, Chang Liu, Xiaoyong Du, Wei Lu, and Anqun Pan. 2021. Systematic Definition and Classification of Data Anomalies in DBMS. *Journal of Software* 33, 3 (2021), 909–930.

[49] Keqiang Li, Siyang Weng, Peiyuan Liu, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, Jianghang Lou, Gui Huang, Weining Qian, and Aoying Zhou. 2023. Leopard: A Black-Box Approach for Efficiently Verifying Various Isolation Levels. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 722–735.

[50] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. Article 143, 12 pages.

[51] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1140–1152.

[52] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 211:1–211:30.

[53] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 667–682.

[54] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 618–622.

[55] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2072–2084.

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

[57] Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. *Proceedings of the VLDB Endowment (VLDB)* (2025).

[58] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 63–80.

[59] Lei Tang, Wensheng Dou, Yingying Zheng, Lijie Xu, Wei Wang, Jun Wei, and Tao Huang. 2025. Proving Cypher Query Equivalence. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*.

[60] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[61] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE Demo)*. 136–140.

[62] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 654–671.

[63] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Yu Gao, Jiansen Song, Liang Xu, Jiaxin Zhu, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Testing Gremlin-Based Graph Database Systems via Query Disassembling. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1695–1707.

[64] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Jiansen Song, Ziyue Cheng, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Differential Optimization Testing of Gremlin-Based Graph Database Systems. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 25–36.

[65] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 302–313.

[66] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 58–71.

Received 2025-02-20; accepted 2025-03-31