Understanding Transaction Bugs in Database Systems

Ziyu Cui[1,2], Wensheng Dou[1,2,3,4*], Yu Gao[1,2], Dong Wang[1,2], Jiansen Song[1,2], Yingying Zheng[1,2], Tao Wang[1,2], Rui Yang[1,2], Kang Xu[3,4], Yixin Hu[5], Jun Wei[1,2,3,4*], Tao Huang[1,2]
[1]State Key Lab of Computer Science at ISCAS, [2]University of CAS, Beijing, China
[3]Nanjing Institute of Software Technology, [4]University of CAS, Nanjing, [5]Sun Yat-sen University, Guangzhou, China
{cuiziyu20,wsdou,gaoyu15,wangdong18,songjiansen20,zhengyingying14,wangtao19,yangrui22}@otcaix.iscas.ac.cn
xukang21@mails.ucas.ac.cn,huyx75@mail2.sysu.edu.cn,{wj,tao}@otcaix.iscas.ac.cn

ABSTRACT

Transactions are used to guarantee data consistency and integrity in Database Management Systems (DBMSs), and have become an indispensable component in DBMSs. However, faulty designs and implementations of DBMSs' transaction processing mechanisms can introduce transaction bugs, and lead to severe consequences, e.g., incorrect database states and DBMS crashes. An in-depth understanding of real-world transaction bugs can significantly promote effective techniques in combating transaction bugs in DBMSs.

In this paper, we conduct the first comprehensive study on 140 transaction bugs collected from six widely-used DBMSs, i.e., MySQL, PostgreSQL, SQLite, MariaDB, CockroachDB, and TiDB. We investigate these bugs from their bug manifestations, root causes, bug impacts and bug fixing. Our study reveals many interesting findings and provides useful guidance for transaction bug detection, testing, and verification.

CCS CONCEPTS

• General and reference → Empirical studies; **• Information systems → Database transaction processing**.

KEYWORDS

Database system, transaction bug, empirical study

ACM Reference Format:
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 *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639207

1 INTRODUCTION

Database Management Systems (DBMSs), e.g., MySQL [11], PostgreSQL [14], SQLite [15], MariaDB [9], CockroachDB [5], and TiDB [19], have been developed for decades. DBMSs adopt Structured Query Language (SQL) to retrieve and manipulate data in databases, and have become an indispensable component for many applications in different domains, e.g., government affairs and electronic shopping.

In DBMSs, multiple users can concurrently retrieve and manipulate data in a database. DBMSs leverage transactions to maintain data consistency and integrity. A transaction usually starts with a `BEGIN` statement, accesses the data in a database with a group of SQL statements, and ends with a `COMMIT` or `ROLLBACK` statement. DBMSs ensure that all statements in a transaction should be executed as a whole, even if DBMSs fail. Ideally, concurrent transactions should be executed in isolation. However, stronger isolation can degrade DBMSs' performance more. To balance consistency and performance, modern DBMSs usually support multiple isolation levels [1, 30, 31, 35–37, 41, 43], e.g., `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE` in MySQL and PostgreSQL [20, 21].

To support the above transaction features, DBMS developers have designed various complex mechanisms, e.g., multi-version concurrency control [38, 63, 69], and optimistic concurrency control [57, 75]. However, faulty designs and implementations of transaction processing mechanisms can introduce a special kind of DBMS bugs, i.e., transaction bugs, which can cause DBMSs to violate their transaction semantics, e.g., their claimed ACID (Atomicity, Consistency, Isolation, and Durability) properties [1, 35–37, 40, 41, 43]. We call transaction bugs in DBMSs as *TXBugs* for short.

TXBugs can cause severe consequences in DBMSs, and easily go unnoticed by DBMS developers. For example, we find that about a third of TXBugs can result in incorrect query results, database states and DBMS states. Considering that DBMSs are usually used to handle important data assets, it is critically important to better understand and detect TXBugs.

Some approaches have been proposed to verify and test transaction processing mechanisms in DBMSs [8, 39, 44, 45, 52, 56, 71]. Transaction verification approaches [8, 39, 56, 71] verify whether transactions' concurrent executions violate the claimed isolation constraints. These approaches can only utilize simple data structures, e.g., *key–value*, and cannot support the rich and complex features in modern DBMSs, e.g., partition configurations for data sharding. For transaction testing, DT^2 [44] utilizes differential testing to detect TXBugs, which cannot handle DBMS-specific features. Troc [45] detects TXBugs by decoupling a pair of transactions into independent statements and building test oracles for transactions. However, it is unclear how effective existing approaches are in detecting real-world TXBugs, and what TXBugs cannot be detected by existing approaches. We also lack a TXBug dataset for TXBug

[*] Wensheng Dou and Jun Wei are the corresponding authors. CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software Chinese Academy of Sciences.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639207

research. We believe that an in-depth study of TXBugs and a TXBug dataset can greatly promote the reliability research in DBMSs' transaction processing mechanisms.

In this paper, we conduct the first comprehensive study on 140 TXBugs collected from six widely-used real-world relational DBMSs, i.e., MySQL [11], PostgreSQL [14], SQLite [15], MariaDB [9], CockroachDB [5], and TiDB [19]. We thoroughly analyze these TXBugs and try to answer the following five research questions:

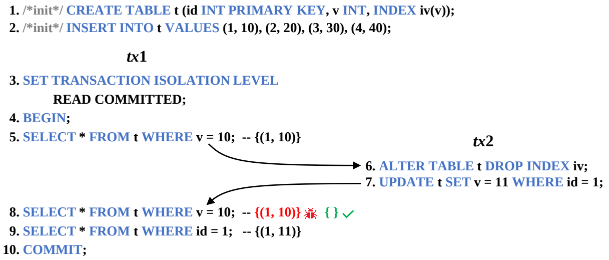
- **RQ1 (Bug manifestation):** How do TXBugs manifest themselves? How are TXBugs triggered?
- **RQ2 (Root cause):** What are root causes for TXBugs?
- **RQ3 (Bug impact):** What impacts do TXBugs have? Can they lead to silent failures?
- **RQ4 (Bug fixing):** How are TXBugs fixed?
- **RQ5 (Detection capability of existing approaches):** How effectively can existing approaches detect TXBugs?

Through our in-depth analysis from the above aspects, we obtain many interesting findings. We summarize some main findings and key lessons as follows.

- Almost all (93.6%) TXBugs do not require more than 3 transactions. Most (84.7%) transactions contain no more than 4 statements. Most (89.3%) TXBugs do not require more than 1 initial table, and most (86.8%) initial tables do not contain more than 5 rows of data. This indicates that TXBugs usually follow the small scope hypothesis [50], and can be effectively detected with small transaction test cases.
- Almost all (94.3%) TXBugs can be triggered deterministically by executing SQL statements in their transaction test cases in a certain order. This indicates that we can detect TXBugs in relatively simple test scenarios without considering nondeterminism in DBMSs.
- Our studied TXBugs violate five kinds of transaction semantics, i.e., atomicity, consistency, isolation, read-only constraint, and statement correctness under transactions. However, existing works mainly focus on isolation violations. This suggests that we also need to explore other kinds of transaction semantics violations.
- Only a small fraction (23.6%) of TXBugs cause explicit failures (e.g., DBMS crashes), while most (76.4%) TXBugs result in silent failures, e.g., incorrect database states and query results. This suggests that we need to design new test oracles to effectively detect silent TXBugs.
- Less than half (45.7%) of TXBugs can be detected by existing approaches [8, 39, 44, 45, 56, 71]. This can draw our community's attention to urgently develop effective TXBug testing and detection approaches.

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

- We present the first empirical study on transaction bugs in real-world DBMSs from five aspects, i.e., bug manifestations, root causes, bug impacts, bug fixing, and detection capability of existing approaches. Our findings can open up new research directions in combating transaction bugs.
- Our 140 documented transaction bugs can serve as a bug benchmark for future work on combating transaction bugs in DBMSs. We have made our collected TXBugs and analysis results available at https://doi.org/10.5281/zenodo.10460740.



```
 1. /*init*/ CREATE TABLE t (id INT PRIMARY KEY, v INT, INDEX iv(v));
 2. /*init*/ INSERT INTO t VALUES (1, 10), (2, 20), (3, 30), (4, 40);

                    tx1

 3. SET TRANSACTION ISOLATION LEVEL
        READ COMMITTED;
 4. BEGIN;
 5. SELECT * FROM t WHERE v = 10;  -- {(1, 10)}                        tx2
                                                      6. ALTER TABLE t DROP INDEX iv;
                                                      7. UPDATE t SET v = 11 WHERE id = 1;

 8. SELECT * FROM t WHERE v = 10;  -- {(1, 10)} 🐞 { } ✓
 9. SELECT * FROM t WHERE id = 1;  -- {(1, 11)}
10. COMMIT;
```

Figure 1: A transaction test case that triggers TXBug TiDB#21498 in TiDB.

2 PRELIMINARIES

We use a real-world TXBug to introduce the basic concepts used in this paper. Figure 1 shows a transaction test case that triggers TXBug TiDB#21498 at the READ COMMITTED isolation level in TiDB. In this transaction test case, we first create a table t (Line 1) and then insert 4 rows of data into table t (Line 2). Note that we create an index iv on the second column v in table t.

We start two concurrent transactions, i.e., $tx1$ and $tx2$. We first set the isolation level as READ COMMITTED for $tx1$ (Line 3), so that $tx1$ can read other transactions' committed data. Then, we start an explicit transaction (Line 4), and execute a SELECT statement at Line 5, which correctly returns {(1, 10)} by TiDB. Then, we execute two statements, i.e., dropping index iv (Line 6) and updating row (1, 10) to (1, 11) (Line 7) in an auto transaction $tx2$. Note that although there is no BEGIN statement in $tx2$, two statements at Line 6–7 are executed as two independent transactions, since each statement in $tx2$ is treated as an *autocommit* transaction by default in TiDB. So, the modifications made by Line 6–7 have been committed to table t after executing Line 7. Since $tx1$ is executed at the READ COMMITTED isolation level, the SELECT statement at Line 8 should retrieve the newest data in table t, and returns an empty query result. However, the SELECT statement at Line 8 incorrectly returns {(1, 10)}, which is the old value of the row with $id = 1$ in table t, and has been modified by Line 7. Note that the SELECT statement at Line 9 correctly returns its query result {(1, 11)}.

For this TXBug, TiDB violates its claimed isolation level, i.e., READ COMMITTED, and returns an incorrect query result for the SELECT statement at Line 8. This TXBug is caused by incorrectly using the table schema for the SELECT statement at Line 8 in TiDB. In TiDB, a transaction at READ COMMITTED isolation level always uses the table schema at its start timestamp. If the table schema is changed while the transaction is executing, the transaction may access data using the unmatched table schema.

We can see that a transaction test case that triggers a TXBug usually consists of two parts, i.e., an initialization stage for creating an initial database, and a testing stage that creates a group of transactions and executes SQL statements in them. For example, in Figure 1, two SQL statements at Line 1–2 create an initial table t and insert 4 rows of data at the initialization stage. Two transactions, i.e., $tx1$ (Line 3–5 and Line 8–10) and $tx2$ (Line 6–7), execute their corresponding statements on table t in a certain order.

DBMSs usually adopt different approaches to create and manage transactions. For example, MySQL and MariaDB support both

Table 1: Target DBMSs in Our Study

DBMS	Release	DB-Engines Ranking	Stars	Database Type	Transaction Type	Isolation Level	Concurrency Control
MySQL	1995	2	9.3K	Traditional	Explicit, XA, auto	RU, RC, RR, SER	Pessimistic
PostgreSQL	1996	4	12.8K	Traditional	Explicit, auto	RC, RR, SER	Pessimistic, Optimistic
SQLite	2000	10	4.3K	Embedded	Explicit, auto	RU, SER	Pessimistic
MariaDB	2009	13	4.9K	Traditional	Explicit, XA, auto	RU, RC, RR, SER	Pessimistic
CockroachDB	2015	60	27.5K	NewSQL	Explicit, auto	SER	Optimistic
TiDB	2017	107	34.5K	NewSQL	Explicit, auto	RC, RR	Pessimistic, Optimistic

explicit transactions and XA transactions [28, 29]. To make our presentation consistent, we define three kinds of transactions, which unify transactions used in our target DBMSs in Section 3.1.

Explicit transaction. An explicit transaction explicitly starts with a `BEGIN` statement, and ends with a `COMMIT` or `ROLLBACK` statement, e.g., $tx1$ in Figure 1. Note that `SET AUTOCOMMIT = 0` statement can implicitly start a new transaction (i.e., similar to a `BEGIN` statement). Thus, we also consider a transaction that starts with `SET AUTOCOMMIT = 0` as an explicit transaction.

XA transaction. A XA transaction starts with `XA START` statement and ends with `XA END` statement. XA transactions are used for processing distributed transactions in MySQL and MariaDB.

Auto transaction. If the *autocommit* mode is enabled, each statement that does not belong to an explicit or XA transaction is implicitly executed as an independent transaction. For easy presentation, we group all consecutive autocommit statements into an auto transaction. For example, $tx2$ in Figure 1 shows an auto transaction, which contains two autocommit statements.

3 METHODOLOGY

3.1 Target DBMSs

We collect TXBugs from six popular DBMSs, i.e., MySQL [11], PostgreSQL [14], SQLite [15], MariaDB [9], CockroachDB [5], and TiDB [19]. Table 1 shows the statistics about our studied DBMSs. MySQL, PostgreSQL, SQLite, and MariaDB are mature and well-developed DBMSs, and rank high in the DB-Engines Ranking [7]. CockroachDB and TiDB are developed in recent years, but are popular (ranked as the 3rd and 5th DBMSs in GitHub) in the open source community. MySQL, PostgreSQL and MariaDB are traditional DBMSs, SQLite is an embedded DBMS, and CockroachDB and TiDB are NewSQL distributed DBMSs.

All these DBMSs support explicit transactions and auto transactions, while MySQL and MariaDB further support XA transactions. These DBMSs support different transaction isolation levels, e.g., `Read Uncommitted` (RU), `Read Committed` (RC), `Repeatable Read` (RR) and `Serializable` (SER) [1, 30, 31, 35–37, 41, 43], and use different concurrency control mechanisms, i.e., pessimistic and optimistic transaction modes [57, 75].

3.2 Collecting TXBugs

Our studied DBMSs are all well maintained and have publicly available issue repositories. MySQL and SQLite maintain their own issue repository websites [12, 16]. PostgreSQL manages its issues by mailing lists [13]. MariaDB manages its issues on JIRA [10], while CockroachDB and TiDB manage their issues on GitHub [6, 18].

We collect TXBugs from these DBMSs' issue repositories. These DBMSs have been developed for a long period (e.g., 27 years for MySQL), and usually contain a large amount of issues, e.g., 109,580 issues in MySQL, and 13,716 issues in TiDB. It is time-consuming and daunting to manually inspect all these issues and identify TXBugs from them. Therefore, we apply some filtering rules to identify relevant issues. First, to keep our study results up-to-date, we focus on the issues confirmed by DBMS developers in the last 5 years, i.e., from January 2018 to December 2022. Second, DBMS developers usually do not label whether an issue is related to TXBugs. Therefore, we use the following keywords, i.e., "transaction", "abort", "commit", "isolation level", and their variations, to retrieve potentially relevant issues in these DBMSs. This search returns us with 7,775 issues. Third, we manually inspect the bug description and developer comments for each retrieved issue, and check whether it is related to transaction processing mechanisms in DBMSs. Note that our search keywords are commonly used in DBMSs. We think that TXBugs should contain at least one of these keywords. We also observe that, although many issues contain our search keywords, they are unrelated to transaction processing mechanisms. In our study, 7,220 (92.9%) issues do not involve transaction processing mechanisms. We exclude these issues from our study.

For each remaining issue, we carefully investigate its description, embedded test cases, developer discussions and available fixing patches, and further rebuild its bug test case step by step. We keep an issue as a TXBug if it needs at least one explicit transaction or XA transaction to trigger, or it needs at least two transactions to trigger. If a bug can be triggered by only one auto transaction, we do not consider it as a TXBug, since it usually belongs to transaction-unrelated bugs, e.g., logic bugs in single `SELECT` statements [64–66]. To maintain the accuracy of our study results, we only keep TXBugs that we can completely understand.

We finally collect 140 TXBugs from our studied DBMSs. Table 2 shows the detailed statistics from their isolation levels and concurrency control mechanisms. We can see that these TXBugs cover various isolation levels and concurrency control mechanisms.

3.3 Analyzing TXBugs

To answer our five research questions, we perform an in-depth analysis on these 140 TXBugs based on their issue descriptions, embedded test cases, developer discussions, and available fixing patches, and further assign them into different categories according to bug manifestations (Section 4), root causes (Section 5), bug impacts (Section 6) and bug fixing (Section 7). Besides, we check whether each TXBug can be detected by existing approaches [8, 39, 44, 45, 56, 71], and analyze their detection capability (Section 8).

Table 2: Collected TXBugs

DBMS	Total	Isolation Levels					Concurrency Control		
		RU	RC	RR	SER	Unspecified	Pessimistic	Optimistic	Unspecified
MySQL	33	1	5	6	1	21	33	-	0
PostgreSQL	6	-	0	0	3	3	0	3	3
SQLite	6	0	-	-	1	5	6	-	0
MariaDB	24	1	3	5	2	17	24	-	0
CockroachDB	9	-	-	-	9	0	-	9	0
TiDB	62	-	9	10	-	49	60	3	0
Total	**140**	**2**	**17**	**21**	**16**	**95**	**123**	**15**	**3**

-: The feature is not supported by the corresponding DBMS.

We adopt the open card sorting approach to build the categories for TXBugs from different dimensions. For each TXBug, we extract necessary information from its issue report, e.g., test cases, expected execution results, developer discussions, and available fixing patches, and then write down its test case step by step. We also reproduce 63 TXBugs for deeply understanding them. For TXBugs that contain fixing patches, we look through their patches to retrieve fixing related information. For other TXBugs that are marked as "Fixed" but do not have patches, we compare their code in the buggy version and fixed version to obtain their patches if possible. To figure out whether a TXBug can be detected by existing approaches [8, 39, 44, 45, 56, 71], we carefully study these approaches, and check whether these approaches can support a TXBug's triggering conditions and design a proper oracle to identify the TXBug, theoretically.

For each TXBug, we try to assign it into an existing category in a dimension (e.g., root cause and bug impact) according to the extracted information. If we cannot find a category for assigning a TXBug, we create a new category. We also refine categories if necessary. All TXBugs are reviewed multiple rounds to ensure correctness and consistency.

3.4 Threats to Validity

First, we collect TXBugs from six widely-used DBMSs. These DBMSs cover representative and important DBMSs, and adopt various transaction processing mechanisms. Our studied TXBugs are collected from six DBMSs without bias. We have not intentionally ignored any TXBugs in these DBMSs. We believe that our studied TXBugs provide a representative sample of TXBugs in these DBMSs.

Second, as an empirical and qualitative bug study, our study results are dependent on the involved researchers' understanding. This may introduce implicit bias towards the expertise of individual researchers. We take several measures to mitigate this threat. (1) We adopt widely-used empirical bug analysis protocols in existing studies [32, 33, 46, 48, 53, 58, 62, 70], e.g., how to collect and analyze bugs. (2) All our studied TXBugs have been independently investigated and discussed by at least three authors. (3) If we have disagreements about a TXBug, e.g., conflicting categories for triggering conditions, we reinvestigate it until we reach a consensus. (4) If we cannot fully understand a TXBug or reach a consensus for a TXBug, we do not take it into consideration. This is a sacrifice that we have to make to maintain the accuracy of our study results.

Third, the replicability and reproducibility of empirical bug studies are common and recognized limitations. To mitigate this threat, we will make our studied TXBugs and detailed study results publicly available. Thus, other researchers should be able to validate our study results easily.

Fourth, TXBugs in other DBMSs, e.g., graph database systems [23, 25, 26] and key-value database systems [24, 27] are not included in our study. Our study may not generalize to those systems not covered in our dataset, because each database system has a unique purpose, design, and implementation. Readers need to be cautious when extending our findings to other database systems.

3.5 TXBug Dataset

Our TXBug dataset contains in total 150 classification labels and around 1400 lines of clear and concise bug descriptions including transaction test cases, bug manifestations, root causes, bug impacts and fixes. The transaction test cases include initial database building statements, transaction statements, and the statement execution order for triggering TXBugs.

We believe that our TXBug dataset can be used as a rich 'bug benchmark' for researchers to combat TXBugs in DBMSs. They will have sample TXBugs to start with and advance their work without having to repeat our multiple-person-month effort.

4 BUG MANIFESTATION

We study TXBugs' triggering conditions at the initialization stage in their transaction test cases (Section 4.1), involved transactions (Section 4.2) and statements (Section 4.3). Finally, we discuss whether TXBugs can be triggered deterministically (Section 4.4).

4.1 Initialization

To trigger a TXBug, transactions in its transaction test case need to run on a specific database. Therefore, we need to build a database at the initialization stage through creating tables, inserting data, etc. We measure the complexity of the initialization stage in TXBugs' transaction test cases from three aspects, i.e., initial tables, initial data, and database properties.

Initial tables. Initial tables are created by `CREATE TABLE` statements at the initialization stage in a TXBug's transaction test case. For tables created by transactions in the transaction test case, we do not count them as initial tables. Note that if the transaction test case does not contain `CREATE TABLE` statements, i.e., the tables used by the transaction test case are not explicitly created at the initialization stage, but directly used in their transactions (e.g., MySQL#92558), we also consider them as initial tables.

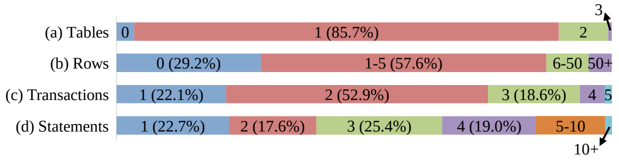


Figure 2: Distributions of TXBugs' triggering conditions.

In total, 140 TXBugs create 151 initial tables. Figure 2a shows the distribution of initial tables for these TXBugs. 120 (85.7%) TXBugs require one initial table, 14 (10.0%) TXBugs require two initial tables, and one TXBug requires three initial tables. Note that 5 TXBugs do not need any table at the initialization stage, e.g., 2 TXBugs execute transactions on built-in tables in the corresponding DBMSs (e.g., built-in table 'information_schema.innodb_trx' in MySQL), and the remaining 3 TXBugs (e.g., PostgreSQL#16771) use tables that are created by transactions in their transaction test cases.

> **Finding 1**: *Most (89.3%) TXBugs require no more than one initial table, and all TXBugs require at most three initial tables.*

Initial data. Initial data is inserted into the initial tables by INSERT statements at the initialization stage in a TXBug's transaction test case. Similar to initial tables, we count the data that is not explicitly inserted at the initialization stage but returned by query statements in transactions, but do not count the data inserted by transactions. Specifically, the concrete data size depends on the number of rows returned by query statements. For those query statements that only return partial data filtering by a WHERE clause, we only count the returned parts as initial data. Take MySQL#92558 as an example. In its initialization stage, no statements insert any data. But, a SELECT count(*) statement in its transaction returns 3 rows of data from its initial table. Thus, we consider that there are 3 rows of initial data.

Figure 2b shows the distribution of initial data in initial tables created by TXBugs' transaction test cases. In Figure 2b, $y(x\%)$ denotes that $x\%$ of initial tables contain y rows of data. Among the 151 initial tables created by 140 TXBugs, 44 (29.2%) initial tables do not contain any data, 87 (57.6%) initial tables contain no more than 5 rows of data, 13 (8.6%) initial tables contain 6 to 50 rows of data, and 7 (4.6%) initial tables contain more than 50 rows of data. It is worth mentioning that MySQL#90980 requires 3 initial tables, among which 2 tables do not have any data, but the remaining one is inserted with more than 4 millions rows of data.

> **Finding 2**: *Most (86.8%) initial tables in TXBugs contain a small amount of data, i.e., no more than 5 rows of data.*

Database properties. To trigger a TXBug, its transaction test case usually requires specific database properties on its initial tables. We discuss database properties from three aspects, i.e., schema properties, table configurations and specific table types. In total, 100 (71.4%) TXBugs require at least one of these database properties. We describe these database properties as follows.

Schema properties in initial tables include three aspects. 86 (61.4%) TXBugs require key constraints (e.g., primary key, unique key, and foreign key), 26 (18.6%) TXBugs require index settings, and 46 (32.9%) TXBugs require column constraints. Figure 1 shows an example of creating an initial table with a primary key on column *id* and an index *iv* on column v (Line 1). Column constraints can be added by keywords, e.g., NOT NULL, DEFAULT NULL and AUTO_INCREMENT. Note that a TXBug may require more than one aspect of the above schema properties.

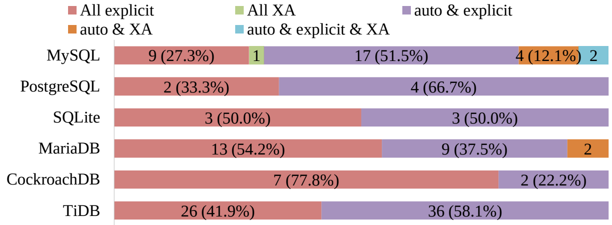


Figure 3: The types of transactions in TXBugs.

Table configurations include two aspects. 15 (10.7%) TXBugs require encoding, and 9 (6.4%) TXBugs require partition. Encoding configurations specify the character set and collation way used by initial tables, which usually differ in different DBMSs. For example, MySQL uses CHARSET keyword [4] while PostgreSQL uses ENCODING keyword [3]. Partition configurations affect how initial tables are stored in DBMSs, e.g., range and list partitioning.

Specific table types contain temporary tables, virtual tables and system tables. Among them, temporary tables and virtual tables can be created by CREATE TEMPORARY TABLE and CREATE VIRTUAL TABLE statements, respectively, while system tables are built-in tables in DBMSs and used to store metadata for DBMSs. Among all collected TXBugs, 7 (5.0%) TXBugs require temporary tables or virtual tables (e.g., SQLite#167b2aac34) and 2 (1.4%) TXBugs execute transactions on system tables (e.g., TiDB#23380).

> **Finding 3**: *Most (71.4%) TXBugs have specific requirements for database properties, e.g., key constraints, column constraints, partition configurations, and virtual tables.*

4.2 Transactions

The distribution of transactions. Transactions are used to trigger TXBugs in their transaction test cases. Figure 2c shows the distribution of transactions for our studied TXBugs. Among these TXBugs, 31 (22.1%) TXBugs only require one transaction, 74 (52.9%) and 26 (18.6%) TXBugs require two and three transactions, respectively. 7 (5.0%) and 2 (1.4%) TXBugs require four and five transactions, respectively. We observe that all our studied TXBugs need no more than five transactions. This indicates that we can trigger TXBugs through a small number of transactions.

The types of transactions. In total, 140 TXBugs create 295 transactions, including 188 (63.7%) explicit transactions, 12 (4.1%) XA transactions, and 95 (32.2%) auto transactions. Figure 3 shows the distribution of the types of transactions for TXBugs in each DBMS. Among them, 61 (43.6%) TXBugs are triggered by all explicit transactions (42.9%) or all XA transactions (0.7%). The remaining 79 (56.4%) TXBugs require the combination of more than one type of transactions. Specifically, 50.7% of TXBugs require the combination of auto transactions and explicit transactions, 4.3% of TXBugs require the combination of auto transactions and XA transactions, and 1.4% of TXBugs require all the three types of transactions.

> **Finding 4**: *Almost all (93.6%) TXBugs require no more than three transactions, and no more than five transactions can trigger all our studied TXBugs. More than half (56.4%) of TXBugs require multiple types of transactions.*

4.3 SQL Statements

The distribution of statements. All the studied TXBugs involve 295 transactions in total. Figure 2d shows the distribution of statements in these transactions. As shown in Figure 2d, 250 (84.7%) transactions contain no more than 4 statements, and 41 (13.9%) transactions contain 5 to 10 statements. Only 4 (1.4%) transactions contain more than 10 statements. For example, an explicit transaction in MySQL#89272 contains 10002 statements, including a `BEGIN` statement, a `COMMIT` statement, and 10000 `INSERT` statements.

Statement types. Except for transaction start and end statements (e.g., `BEGIN` and `COMMIT`), various statements are involved in TXBugs. `SELECT` statements (48.6%) are involved most often, followed by `INSERT` statements (45.7%) and `UPDATE` statements (40.0%), and `DELETE` statements (12.9%). In addition, many TXBugs also involve other types of statements, e.g., `ALTER` (17.1%, e.g., SQLite#596d059af7), `SHOW` (13.6%, e.g., MySQL#104245), `SET` (9.3%, e.g., MariaDB#16024), `ADMIN` (2.9%, e.g., TiDB#20910), and `RESET` (0.7%, e.g., PostgreSQL#17385).

> **Finding 5**: *Most (84.7%) transactions contain no more than 4 statements. TXBugs usually involve varied types of statements, e.g.,* `SELECT`, `INSERT`, `ALTER`, `SET`, `ADMIN`, *and* `RESET`.

4.4 Bug Determinism

We use the following criteria to determine whether a TXBug can be triggered deterministically. If the issue report of a TXBug explicitly states that the bug is triggered by chance, we consider that this bug cannot be triggered deterministically. If the issue report describes the transaction test case that triggers a TXBug clearly, and DBMS developers do not complain that the bug cannot be reproduced, we consider that this bug can be triggered deterministically.

Surprisingly, we find that 132 (94.3%) TXBugs can be triggered deterministically. That said, we can trigger these 132 TXBugs by executing the statements in their transaction test cases on the target DBMS in a certain order, regardless of DBMS's nondeterminism. Figure 1 shows such a deterministic TXBug.

The remaining 8 (5.7%) TXBugs cannot be triggered deterministically. Listing 1 shows nondeterministic TXBug MySQL#101667 triggered by repeatably executing the two transactions in parallel. In this transaction test case, we first set InnoDB adaptive hash index to be ON (Line 1), create a table t with a column id (Line 2), and insert one row of data (Line 3). Then, we start two explicit transactions $tx1$ and $tx2$ in parallel. Transaction $tx1$ retrieves the row whose column id is 'foo' (Line 4–6). Transaction $tx2$ deletes the row whose column id is 'foo' and then aborts itself (Line 7–9). Since $tx2$ never commits (Line 9), Line 5–6 in $tx1$ should always return a query result {('foo')}. However, when repeatably running $tx1$ and $tx2$ in parallel, the `SELECT` statement at Line 6 in $tx1$ can incorrectly return an empty query result accidentally.

```
1. /*init*/ SET innodb_adaptive_hash_index = ON;
2. /*init*/ CREATE TABLE t (id VARCHAR(16) NOT NULL
     PRIMARY KEY);
3. /*init*/ INSERT INTO t VALUES('foo');
4. /*tx1*/ SET AUTOCOMMIT = 0;
5. /*tx1*/ SELECT * FROM t WHERE id = 'foo';
6. /*tx1*/ SELECT * FROM t WHERE id = 'foo';
   -- Accidentally return an empty result ✘
   -- {(`foo')} ✔
7. /*tx2*/ SET AUTOCOMMIT = 0;
8. /*tx2*/ DELETE FROM t WHERE id = 'foo';
9. /*tx2*/ ROLLBACK;
```

Listing 1: Nondeterministic TXBug MySQL#101667

For the 8 nondeterministic TXBugs, 5 TXBugs can be triggered by repeatably executing their statements or transactions, e.g., Listing 1. The remaining 3 TXBugs can be triggered by manually controlling the target DBMS to reach a specific DBMS state. For example, in TiDB#23363, we need to use TiDB's built-in transaction mechanism async-commit to control the transaction commit [2].

> **Finding 6**: *Almost all (94.3%) TXBugs can be triggered deterministically. For nondeterministic TXBugs, we need to adopt some new triggering strategies, e.g., repeatably executing transactions, and manually controlling database states.*

5 ROOT CAUSE

In TXBugs, faulty designs and implementations violate DBMSs' transaction semantics. In this section, we analyze TXBugs' root causes from the perspective of the violated transaction semantics. Table 3 shows the categories for the violated transaction semantics. Specifically, we obtain these categories by considering transactions' ACID properties, the semantics of read-only transactions and the correctness of statements under transactions.

```
1. /*init*/ CREATE TABLE t (c1 INT);
2. /*init*/ INSERT INTO t VALUES(0);
3. /*tx1*/ SET AUTOCOMMIT = ON;
4. /*tx1*/ BEGIN;
5. /*tx1*/ UPDATE t SET c1 = c1 + 1;
6. /*tx1*/ SET AUTOCOMMIT = ON;
7. /*tx1*/ ROLLBACK; -- fails to revert database state
8. /*tx2*/ SELECT * FROM t; -- {(1)} ✘  {(0)} ✔
```

Listing 2: TiDB#36581 violates atomicity

Atomicity violations (2 TXBugs). In DBMSs, statements in a transaction should be executed as a whole, i.e., either a transaction succeeds completely or fails completely. For a TXBug, if not all statements in a committed transaction succeed, or if the modifications of a rollback transaction are not reverted, we categorize the TXBug into atomicity violation. Listing 2 shows TXBug TiDB#36581 caused by atomicity violation. In this transaction test case, an explicit transaction $tx1$ updates the record 0 in table t to 1 (Line 5), executes `SET AUTOCOMMIT = ON` (Line 6) and rolls back (Line 7). The database state is expected to be reverted to the state before executing $tx1$, i.e., table t should contain {(0)}. However, table t contains {(1)} (Line 8), thus violating transaction atomicity.

Table 3: Transaction Semantic Violations

Violation		# TXBugs
Atomicity		2 (1.4%)
Consistency	DBMS state	31 (22.1%)
	Application state	11 (7.9%)
	Database constraint	3 (2.1%)
Isolation	Insufficient isolation	27 (19.3%)
	Excessive isolation	17 (12.2%)
Read-only constraint		4 (2.9%)
Statement correctness		45 (32.1%)

Consistency violations (45 TXBugs). Consistency ensures that transactions only make changes in the predefined and predictable ways, and do not create unintended consequences for the integrity of the data in DBMSs. Otherwise, a consistency violation occurs. We summarize consistency violations into three subcategories according to the types of data that are corrupted.

- Consistency violations of DBMS states (31 TXBugs). DBMSs usually maintain some execution information in built-in tables, e.g., transaction and locking information. These information can be wrongly recorded or corrupted during transaction executions. For example, in TiDB#23380, a transaction's start time should have been saved in TiDB's built-in table 'information_schema.processlist', but it was not.
- Consistency violations of application states (11 TXBugs). In these violations, the statements in transactions are not executed in a predefined way, and break the data integrity expected by applications. Listing 3 shows TXBug TiDB#19585 that breaks the integrity of application states. In this transaction test case, table t is required to be stored in partition. An explicit transaction $tx1$ first inserts a value 10 (Line 4), then tries to update the value 1 to 11 (Line 5). After $tx1$ completes, the database state is expected to be {(10), (11)} (Line 7). However, incorrectly updating the value 1 twice at Line 5 creates a corrupted application state {(10), (21)}.
- Consistency violations of database constraints (3 TXBugs). In these violations, a transaction execution can break the constraints defined in databases, e.g., primary key and unique key constraints. For example, in MySQL#101706, the execution of two concurrent transactions inserts two same records into a table that has a unique key constraint.

```
1. /*init*/ CREATE TABLE t (c1 INT PRIMARY KEY)
     PARTITION BY RANGE (c1) (PARTITION p0 VALUES LESS
     THAN (10),PARTITION p1 VALUES LESS THAN MAXVALUE);
2. /*init*/ INSERT INTO t VALUES(1);
3. /*tx1*/ BEGIN;
4. /*tx1*/ INSERT INTO t VALUES(10);
5. /*tx1*/ UPDATE t SET c1=c1+10 WHERE c1 IN (1, 11);
6. /*tx1*/ COMMIT;
7. /*tx2*/ SELECT * FROM t ORDER BY c1;
   -- {(10), (21)} ✘  {(10), (11)} ✔
```

Listing 3: TiDB#19585 violates the application's consistency

Isolation violations (44 TXBugs). The isolation of concurrent transactions ensures that they do not interfere with or affect one another. As shown in Table 1, our studied DBMSs usually support multiple isolation levels [1, 30, 31, 37]. Transaction execution should follow the claimed isolation level. Otherwise, an isolation violation occurs. We divide isolation violations into two sub-categories, i.e., insufficient isolation and excessive isolation.

- Insufficient isolation (27 TXBugs). Concurrent transactions are not sufficiently isolated, and cause weaker isolation than claimed. In Listing 4, the INSERT statement in $tx2$ (Line 6) should be blocked by $tx1$, since there is a write-write conflict. However, the INSERT statement is not blocked due to insufficient isolation.
- Excessive isolation (17 TXBugs). If a statement $stmt$ in a transaction $tx1$ should not been blocked by another transaction $tx2$, but $stmt$ is blocked by $tx2$ to avoid conflict, we consider that the isolation is excessive, e.g., MariaDB#24224.

```
1. /*init*/ CREATE TABLE t (c1 INT, c2 INT, c3 INT,
     PRIMARY KEY(c1, c2));
2. /*init*/ INSERT INTO t VALUES(1, 1, 1);
3. /*tx1*/ BEGIN;
4. /*tx2*/ BEGIN;
5. /*tx1*/ DELETE FROM t WHERE c1 = 1;
6. /*tx2*/ INSERT INTO t VALUES(1, 1, 2);
   -- ERROR: duplicate constraint name: "t2_c2_fkey" ✘
   -- Blocked ✔
```

Listing 4: TiDB#20535 is caused by insufficient isolation

Read-only constraint violations (4 TXBugs). Read-only transactions cannot perform modification operations. If a read-only transaction successfully executes a modification operation that is not allowed to be executed, we consider that the read-only constraint is violated. For example, in TiDB#22658, the read-only transaction should fail to insert values into a table, but successfully executes the INSERT statement.

```
1. /*init*/ CREATE TABLE t1 (c1 INT PRIMARY KEY);
2. /*init*/ CREATE TABLE t2 (c1 INT PRIMARY KEY, c2 INT
      NOT NULL REFERENCES t1);
3. /*tx1*/ BEGIN;
4. /*tx1*/ ALTER TABLE t2 DROP CONSTRAINT t2_c2_fkey;
5. /*tx1*/ ALTER TABLE t2 ADD CONSTRAINT t2_c2_fkey
     FOREIGN KEY (c2) REFERENCES t1 (c1) ON DELETE
     CASCADE;
   -- ERROR: duplicate constraint name: "t2_c2_fkey" ✘
   -- Successfully execute ✔
6. /*tx1*/ COMMIT;
```

Listing 5: CockroachDB#55184 violates statement correctness

Statement correctness violations (45 TXBugs). These TXBugs do not violate the previous four kinds of transaction semantics. However, a statement in the involved transactions cannot be correctly executed under the specific transaction context. Listing 5 shows TXBug CockroachDB#55184 caused by statement correctness violation. In this transaction test case, we create table $t1$ with a primary key on column $c1$, and table $t2$ with a primary key on column $c1$ and a foreign key on column $c2$ (Line 1–2). In transaction $tx1$, we can successfully drop the foreign key constraint (Line 4). But, when we add back the same foreign key constraint, CockroachDB reports an error "duplicate constraint name" (Line 5). However, this error should not be reported, since we have deleted the constraint at Line 4.

> **Finding 7**: *TXBugs violate five kinds of transaction semantics, i.e., atomicity, consistency, isolation, read-only constraint, and statement correctness under transactions.*

Table 4: Failure Symptoms of TXBugs

Bug Observability	Failure Symptom	# TXBugs
Explicit (23.6%)	DBMS error	28 (20.0%)
	DBMS unavailability	5 (3.6%)
Silent (76.4%)	Incorrect DBMS state	16 (11.4%)
	Incorrect database state	10 (7.2%)
	Incorrect query result	21 (15.0%)
	Performance degradation	15 (10.7%)
	Missing blocking	9 (6.4%)
	Incorrect error reporting	36 (25.7%)

6 BUG IMPACT

To better understand how severe TXBugs are, we investigate TXBugs' failure symptoms (Section 6.1) and the observability of TXBugs (Section 6.2). Finally, we discuss the priority of TXBugs (Section 6.3).

6.1 Failure Symptoms

TXBugs can cause DBMSs to violate their transaction semantics, and lead to various severe consequences. Table 4 shows their failure symptoms, including DBMS errors (20.0%), DBMS unavailability (3.6%), incorrect DBMS states (11.4%), incorrect database states (7.2%), incorrect query results (15.0%), performance degradation (10.7%), missing blocking (6.4%), and incorrect error reporting (25.7%).

A *DBMS error* happens when an assertion failure (e.g., null pointer dereference in TiDB#23179) or an operation error (e.g., malformed database disk image in SQLite#745f1abcdc) is thrown. A *DBMS unavailability* happens when the DBMS crashes or hangs forever.

An *incorrect DBMS state* happens when a DBMS's internal state (not user data) is broken. For example, in TiDB#23380, a transaction's start time information in TiDB's built-in table 'information_schema.processlist' is missing. An *incorrect database state* happens when wrong user data is stored in the tested database after executing a TXBug's transaction test case, e.g., Listing 2 and Listing 3. An *incorrect query result* happens when a SELECT statement in a TXBug's transaction test case returns a wrong result. Note that TXBugs that cause incorrect database states can also lead to incorrect query results. We do not count such TXBugs into incorrect query results to avoid double counting.

TXBugs can cause *performance degradation* when unnecessary (or long) locks are acquired by concurrent transactions, e.g., MariaDB#24224.

The transaction test cases in 45 TXBugs do not clearly provide their failure symptoms as above. We classify them into two categories. *Missing blocking* happens when a statement in a transaction should be blocked, but it is successfully executed without being blocked (e.g., TiDB#17851). *Incorrect error reporting* happens when an unexpected error is reported, an expected error is not reported, or an incorrect error message is reported. Note that we do not count TXBugs that cause DBMS errors into incorrect error reporting.

Finding 8: *All TXBugs can cause severe consequences, e.g., DBMS errors, DBMS unavailability, incorrect DBMS states, incorrect database states, and incorrect query results.*

6.2 Silent Failures

Among all studied TXBugs, 33 (23.6%) TXBugs can cause explicit failures, i.e., DBMS errors and DBMS unavailability, which can be used as test oracles to detect TXBugs. However, 107 (76.4%) TXBugs only lead to silent failures, i.e., incorrect DBMS states, incorrect database states, incorrect query results, performance degradation, missing blocking, and incorrect error reporting. For a transaction test case that causes a silent failure, without understanding its execution semantics, we cannot judge whether its execution is correct by inspecting its failure symptom.

Figure 1 shows a silent TXBug, in which the SELECT statement at Line 8 returns an incorrect query result. We can easily overlook this bug since it will not trigger explicit failures, and we cannot know whether its query result is correct without understanding the concrete semantics of the transaction test case.

Finding 9: *Most (76.4%) TXBugs lead to silent failures, which cannot be detected through simply validating their failure symptoms, e.g., incorrect database states and query results.*

6.3 Priority

Developers usually use bug priorities to express the importance of TXBugs. Different DBMSs adopt their own bug priority levels. To facilitate our analysis, we unify their priority levels into three levels, i.e., *critical*, *major* and *minor*.

- In MySQL, we classify S1 (Critical) into *critical*, S2(Serious) into *major*, and S3 (Non-critical), S5 (Performance) and S6 (Debug Builds) into *minor*. We exclude S4 (feature request) and S7 (test cases), since we do not consider them as bugs.
- In MariaDB, we classify Critical into *critical*, Blocker and Major into *major*, and Minor and Trivial into *minor*.
- TiDB contains four priority levels, i.e., Critical, Major, Moderate and Minor. We classify Moderate into *minor*.
- In SQLite, we classify Severe and Critical into *critical*, Important into *major*, and Minor and Cosmetic into *minor*.
- CockroachDB and PostgreSQL developers do not prioritize their bugs, so we do not assign priority levels for TXBugs in them.

We assign the unified priorities for 124 (88.6%) TXBugs. Among them, 33 (23.6%) TXBugs are classified as critical, 55 (39.3%) TXBugs are classified as major, and 36 (25.7%) TXBugs are classified as minor.

Finding 10: *TXBugs are considered severe by developers, and two thirds (62.9%) of TXBugs have critical or major priorities.*

7 BUG FIXING

For the 140 TXBugs, 83 (59.3%) TXBugs have been fixed by DBMS developers, and 57 (40.7%) have not been fixed yet. Out of the 83 fixed TXBugs, 5 TXBugs are fixed by modifying the corresponding DBMS documents only, e.g., MySQL#103672. The remaining 78 TXBugs are fixed by code patches. We do not investigate TXBugs' detailed code fixing strategies, because TXBugs usually involve complex transaction processing mechanisms, and their fixing patches are usually too complex for us (not DBMS developers) to understand.

We extract the fixing patches of these 78 fixed TXBugs by following the methodology in Section 3.3. Finally, we obtain the fixing patches for 70 TXBugs. We cannot find their patches for the remaining 8 TXBugs. For the 70 TXBugs, we further measure their fixing complexity from four aspects, i.e., the number of patches, the number of affected files, lines of code, and the number of days to fix. On average, fixing a TXBug involves 5 patches, 5 files and 150 lines of code change, and takes 102 days.

7.1 Unfixed TXBugs

For the 57 (40.7%) unfixed TXBugs, we investigate why they are not fixed, and how difficult to fix them. Note that these unfixed TXBugs are also important. Among them, 3 TXBugs are classified as critical, 22 TXBugs are classified as major, 21 TXBugs are classified as minor, and the remaining 11 TXBugs do not have priority labels.

Duration. We measure the duration of these unfixed TXBugs from the date when they were reported to January 1, 2023. The average durations for unfixed TXBugs in MySQL, PostgreSQL, MariaDB, CockroachDB, and TiDB are 964 days, 1,046 days, 709 days, 365 days, and 360 days, respectively. Note that all TXBugs in SQLite have been fixed. This indicates that it is challenging to diagnose and fix TXBugs, e.g., MySQL#90987 remains unfixed for 1,691 days.

Unfixed reasons. Developers usually do not provide detailed reasons why they have not fixed some TXBugs. Thus, we carefully read developers' discussions, and extract the reasons for 16 unfixed TXBugs. In 3 unfixed TXBugs, developers encounter difficulty when diagnosing their root causes. For example, in MySQL#104833, developers commented that the root cause is difficult to diagnose: "The exact cause of the bug is not found yet, but we are working on it very intensively". In 5 unfixed TXBugs, developers cannot figure out their correct transaction semantics. For example, in TiDB#21506, developers explained that the bug is unfixed since the expected behavior of the `INSERT SELECT` statement is undefined: "We need to discuss the expected behavior for the insert select statement first". In the remaining 8 unfixed TXBugs, developers cannot figure out their fixing solutions. For example, in CockroachDB#55184, developers directly commented that the bug is hard to fix: "This is currently an inherent limitation of schema changes. We're making improvements in this area, but they're several releases out".

Finding 11: *Fixing TXBugs is challenging. On average, fixing a TXBug requires 150 lines of code and takes 102 days. About a half (40.7%) of TXBugs have not been fixed due to various reasons, e.g., difficult to diagnose and fix, and unclear transaction semantics.*

8 DETECTION CAPABILITY OF EXISTING APPROACHES

We investigate how effectively existing transaction verification and testing approaches [8, 39, 44, 45, 56, 71] can detect TXBugs. Existing approaches adopt random strategies to generate transaction test cases to expose TXBugs. Therefore, if we run an existing approach for a long time, we may still miss a TXBug since the approach may not generate a proper test case to trigger it. This does not indicate that the approach lacks the capability to detect this TXBug. Thus, we theoretically analyze existing approaches' maximum detection capability for our studied TXBugs. Specially, we study these approaches, and check whether they can support TXBugs' triggering conditions and contain proper test oracles to detect these TXBugs. In total, we find that existing approaches can only detect 64 out of 140 (45.7%) TXBugs.

8.1 Transaction Verification

Existing transaction verification approaches [8, 39, 56, 71] detect TXBugs through analyzing transaction execution history and checking whether DBMSs violate their claimed isolation levels. To build a precise dependency graph among transactions [30, 31], these approaches can only support $key-value$ data structures and limited operations, e.g., $read(key)$ and $write(key, value)$. They further analyze the dependency graph, and treat dependency cycles as bugs that violate DBMSs' claimed isolation levels. They can only detect insufficient isolation violations in Section 5.

For a TXBug, if its transaction test case satisfies one of the following conditions, we can safely conclude that it cannot be detected by existing transaction verification approaches. (1) The test case requires different database structures from $key-value$ data structures. (2) The test case involves complex operations other than $read(key)$ and $write(key, value)$. For example, in Figure 1, the `SELECT` statements at Line 5 and 8 cannot be treated as $read(key)$ operations. (3) Its failure symptom cannot be reflected in the transaction execution history (e.g., unnecessary locks and incorrect error reporting), and we cannot identify dependency cycles among its transactions. For example, in Listing 5, there is only one transaction, thus we cannot identify a dependency cycle.

We investigate all 140 TXBugs, and check whether they satisfy the above three conditions. We find that, only 4 TXBugs satisfy transaction verification approaches' preconditions, and can be detected by these approaches.

8.2 Transaction Testing

In contrast to transaction verification approaches, transaction testing approaches [44, 45] are able to support complex database structures and statements.

DT^2 [44] adopts differential testing to detect TXBugs. DT^2 randomly generates concurrent transactions based on transaction features that are commonly supported by multiple DBMSs, submits them to target DBMSs, and compares transaction execution results on multiple DBMSs to identify discrepancies. If a TXBug satisfies one of the following conditions, DT^2 cannot detect it. (1) The TXBug's test case involves DBMS-specific features, e.g., CockroachDB#46276's test case involves CockroachDB-specific statement `UPSERT`, which is a combination of the `UPDATE` and `INSERT` statements. (2) The TXBug's test case has the same incorrect execution behavior on multiple DBMSs, e.g., a transaction test case triggers the same buggy behavior in MariaDB (MariaDB#26642) and TiDB (TiDB#28212).

Troc [45] detects TXBugs by constructing a test oracle for concurrent transactions. Troc decouples a pair of transactions into independent statements, and executes these statements on their specific database views to obtain the expected execution results. Troc compares transactions' actual execution results and expected execution results, and treats any discrepancy as a TXBug. If a TXBug's test case involves features that Troc cannot apply (e.g., more than two transactions, unsupported statements like `ALTER` (e.g., Figure 1), and optimistic transaction mode), or TXBug's failure symptoms cannot be detected by Troc (e.g., unnecessary blocking and performance degradation), Troc cannot detect it.

We investigate all 140 TXBugs, and check whether they satisfy the above two testing approaches' preconditions. We find that only 52 TXBugs satisfy DT^2's preconditions, and can be detected by DT^2. Only 25 TXBugs satisfy Troc' preconditions, and can be detected by Troc. Note that among these 52 TXBugs, 2 TXBugs can also be detected by transaction verification approaches. Among these 25 TXBugs, one TXBug can also be detected by transaction verification approaches, and 15 TXBugs can also be detected by DT^2.

Finding 12: *Less than half (45.7%) of TXBugs can be detected by existing transaction verification and testing approaches.*

9 LESSONS LEARNED

Finding 12 shows that existing transaction verification and testing approaches cannot effectively detect TXBugs. Our community urgently needs to develop more effective approaches for combating TXBugs. As the first empirical study on TXBugs in DBMSs, we believe that our findings can help DBMS developers and researchers to improve DBMSs' reliability. In this section, we discuss lessons learned, implications to existing works, and opportunities for new researches in combating TXBugs.

9.1 Detecting More Types of TXBugs

Finding 7 shows that TXBugs can violate various transaction semantics, e.g., atomicity, consistency, isolation, read-only constraint and statement correctness under transactions. Existing verification approaches [8, 39, 56, 71] mainly focus on detecting insufficient isolation violations, while DT^2 [44] and Troc [45] can further detect some other violations except insufficient isolation violations, e.g., consistency violations. However, these approaches cannot detect some types of our studied TXBugs, e.g., atomicity and read-only constraint violations. Our community urgently needs to invest more effort to combat more types of TXBugs.

9.2 Transaction Testing

Software testing plays an important role in exposing bugs. Although many approaches have been proposed for DBMS testing [17, 34, 49, 51, 54, 55, 59–61, 64–68, 73, 74, 77, 78], there are only few works for transaction testing in DBMSs [44, 45]. Our study provides guidance that can be exploited by transaction testing from several aspects.

Transaction test case generation. A well-designed transaction test case generation approach can greatly improve the effectiveness of exposing TXBugs. We find that transaction test cases that trigger TXBugs are extremely diverse. Transaction test cases should consider specific database properties (Finding 3), multiple types of transactions (Finding 4), and various types of SQL statements (Finding 5). However, existing approaches, e.g., Cobra [71] and Elle [56], can only generate transactions that use simple read(*key*) and write (*key*) operations on simple *key–value* data structures. DT^2 [44] can only generate the common SQL features that are commonly supported by multiple DBMSs. Therefore, we urgently need effective transaction test case generation approaches, which can cover various SQL features revealed by our study.

Our study shows that TXBugs usually follow the *small scope hypothesis* [50], and only require limited amounts of database tables (Finding 1), data (Finding 2), transactions (Finding 4), SQL statements (Finding 5), etc. These findings indicate that we can generate (or enumerate possible) small test cases to significantly reduce the space of transaction testing, and can still effectively detect TXBugs within limited time and resources. Generating large numbers of transactions and data for transaction verification like Cobra [71] and Elle [56] should be inefficient.

Oracle design. Test oracles are the key to effectively reveal TXBugs. For explicit failures caused by TXBugs (e.g., DBMS crashes and errors), researchers can design an unified test oracle to detect TXBugs based on our identified explicit failures in Finding 9. However, Finding 9 also shows that most TXBugs can only cause silent failures. This indicates that researchers need to develop new test oracles to reveal silent TXBugs.

To make silent TXBugs detectable, DT^2 [44] adopts differential testing and compares transaction execution results on multiple DBMSs. Troc [45] focuses on isolation violations and constructs test oracles according to the claimed isolation level. But, these existing approaches' test oracles cannot detect TXBugs that cause performance degradation, DBMS errors, statement correctness violations, etc. Finding 7 shows that in TXBugs, concurrent transactions usually violate certain transaction semantics. Thus, we can build a precise transaction semantics to expose TXBugs.

Concurrent transaction testing. Intuitively, TXBugs can usually be revealed by highly concurrent transaction execution. Thus, existing approaches, e.g., Cobra [71] and Elle [56], try to expose TXBugs through a lot of concurrent transactions. However, Finding 6 shows that almost all TXBugs can be triggered deterministically, and manifest themselves by executing their transaction test cases in certain order. Based on this finding, researchers can design effective concurrent transaction testing approaches, which can more efficiently expose TXBugs.

For nondeterministic TXBugs, we find that we can adopt simple triggering strategies to trigger them in Finding 6. Researches can use these strategies to further expose nondeterministic TXBugs.

9.3 Transaction Verification

Some transaction verification techniques have been proposed to verify whether DBMSs violate their claimed transaction isolation guarantees by analyzing transaction execution history on simple *key–value* data structures in DBMSs, e.g., Elle [56] and Cobra [71]. However, Finding 12 shows that only few TXBugs can be detected by these approaches because they cannot support complex transactions in modern DBMSs. Therefore, we urgently need to develop new transaction verification techniques to record and analyze complex transactions' execution history, e.g., complex database structures (Finding 3), and more types of statements (Finding 5).

9.4 Transaction Semantics

We observe that TXBugs usually violate some transaction semantics, e.g., atomicity, consistency, isolation, and read-only constraint (Finding 7). We find that some TXBugs have not been fixed because their involved transaction semantics are still unclear for DBMS developers (Finding 11). During our study, we also find that the transaction semantics among different DBMSs may be inconsistent. For example, TiDB and MySQL take snapshots for transactions at different timing [22], which can cause different results for the same transactions. These unclear and inconsistent transaction semantics can also cause ambiguity when database application developers migrate their applications among DBMSs. We urgently need a clear and unified specification for transaction semantics, which can greatly benefit the DBMS community.

9.5 TXBug Diagnosis

Our findings (Finding 1, 2, 4, and 5) show that TXBugs usually follow the small scope hypothesis. While existing approaches [8, 39, 56, 71] usually generate large transaction test cases, which may trouble DBMS developers during bug diagnosis. Therefore, new approaches on transaction test case reduction can help DBMS developers to understand TXBugs.

During our study, we find that many different transaction test cases can trigger the same TXBug, and cause duplicate bug reports. For example, MySQL#105030, MySQL#105670, MySQL#107887 and MySQL#104986 trigger the same TXBug. New approaches on automatically detecting duplicate TXBug reports can help DBMS developers to reduce efforts and save time during bug diagnoses.

10 RELATED WORK

We introduce related works that we have not discussed yet.

Empirical bug studies. Empirical bug studies play an important role in improving the reliability of software systems. Many bug studies have been conducted for different kinds of bugs, e.g., concurrency bugs [58, 62, 72], crash / network partition recovery bugs in distributed systems [32, 33, 46], compiler bugs in GCC and LLVM [70], bugs in deep learning systems [42, 47] and bugs in cloud systems [48]. These empirical studies have motivated researchers to develop various techniques to combating related bugs. In our study, we adopt similar study methodologies to existing studies in the process of bug selection, bug analysis, and bug categorization. To the best of our knowledge, our work is the first comprehensive study on TXBugs in DBMSs.

Bug detection in DBMSs. Some techniques have been proposed to combat logic and crash bugs in DBMSs [17, 34, 49, 51, 54, 55, 59–61, 64–68, 73, 74, 76–78]. SQLsmith [17] detects DBMS bugs by generating random SQL statements. RAGS [67] utilizes differential testing to find bugs in DBMSs. SQLancer [64–66] detects logic bugs in single SELECT statements. Squirrel [78] presents a fuzzing framework to find bugs in DBMSs. Grand [77] and DOT [76] present differential testing approaches to find logic bugs in graph database systems (GDBs). These approaches mainly focus on detecting bugs in single SELECT statements, and cannot detect TXBugs in DBMSs.

11 CONCLUSION

Incorrect designs and implementations in DBMSs' transaction processing mechanisms can introduce transaction bugs, which lead to severe consequences. We conduct the first in-depth study on 140 transaction bugs from six popular DBMSs. From our study, we obtain many interesting findings and lessons. We believe that our study can be beneficial for DBMS and SE researchers from many aspects, e.g., transaction testing, verification and semantics.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62302493, 62072444, U20A6003), Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044), Major Program (JD) of Hubei Province (2023BAA018).

REFERENCES

[1] 2022. The ANSI isolation levels. https://renenyffenegger.ch/notes/development/databases/SQL/transaction/isolation-level.
[2] 2022. Async Commit in TiDB. https://docs.pingcap.com/tidb/stable/release-5.0.0#async-commit.
[3] 2022. Character Set Support in PostgreSQL. https://www.postgresql.org/docs/current/multibyte.html.
[4] 2022. Character Sets, Collations, Unicode in MySQL. https://dev.mysql.com/doc/refman/8.0/en/charset.html.
[5] 2022. CockroachDB. https://www.cockroachlabs.com.
[6] 2022. CockroachDB issues. https://github.com/cockroachdb/cockroach/issues/.
[7] 2022. DB-Engines. https://db-engines.com/en/ranking.
[8] 2022. Gretchen: Offline serializability verification, in Clojure. https://jepsen.io/.
[9] 2022. MariaDB. https://mariadb.org.
[10] 2022. MariaDB JIRA issues. https://jira.mariadb.org/issues/.
[11] 2022. MySQL. https://www.mysql.com.
[12] 2022. MySQL Bugs Home. https://bugs.mysql.com/.
[13] 2022. Pgsql-bugs. https://www.postgresql.org/list/pgsql-bugs/.
[14] 2022. PostgreSQL. https://www.postgresql.org.
[15] 2022. SQLite. https://www.sqlite.org/index.html.
[16] 2022. SQLite Ticket Main Menu. https://www.sqlite.org/src/reportlist.
[17] 2022. SQLsmith. https://github.com/anse1/sqlsmith.
[18] 2022. TiDB issues. https://github.com/pingcap/tidb/issues/.
[19] 2022. TiDB, PingCAP. https://pingcap.com.
[20] 2022. Transaction Isolation in PostgreSQL. https://www.postgresql.org/docs/14/transaction-iso.html.
[21] 2022. Transaction Isolation Levels in MySQL. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.
[22] 2022. Transactions in TiDB. https://docs.pingcap.com/tidb/stable/transaction-overview.
[23] 2023. JanusGraph. https://janusgraph.org.
[24] 2023. Memcached. http://www.memcached.org/.
[25] 2023. Neo4j. https://neo4j.com/.
[26] 2023. OrientDB. https://orientdb.org.
[27] 2023. Redis. https://redis.com/.
[28] 2023. XA Transactions in MariaDB. https://mariadb.com/kb/en/xa-transactions/.
[29] 2023. XA Transactions in MySQL. https://dev.mysql.com/doc/refman/8.0/en/xa.html.
[30] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph. D. Dissertation. Massachusetts Institute of Technology.
[31] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of International Conference on Data Engineering (ICDE)*. 67–78.
[32] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 351–368.
[33] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 51–68.
[34] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*. 2060–2071.
[35] 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* 7, 3 (2013), 181–192.
[36] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Transactions on Database Systems* 41, 3 (2016), 15:1–15:45.
[37] 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 International Conference on Management of Data (SIGMOD)*. 1–10.
[38] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[39] 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.

[40] 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.

[41] 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.

[42] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6 (2023), 31 pages.

[43] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 73–82.

[44] 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 International Conference on Automated Software Engineering (ASE)*. 35:1–35:12.

[45] 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 International Conference on Software Engineering (ICSE)*. 1123–1135.

[46] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 539–550.

[47] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *Proceedings of International Conference on Software Engineering (ICSE)*. 147–158.

[48] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*. 1–14.

[49] 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 International Symposium on Software Testing and Analysis (ISSTA)*. 163–174.

[50] Daniel Jackson and Craig A. Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering* 22, 7 (1996), 484–495.

[51] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 4949–4965.

[52] 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.

[53] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 77–88.

[54] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proceedings of the VLDB Endowment (VLDB)* 13, 1 (2019), 57–70.

[55] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 140–149.

[56] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proceedings of the VLDB Endowment* 14, 3 (2020), 268–280.

[57] Hsiang-Tsung Kung and John T Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[58] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 517–530.

[59] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *Proceedings of International Conference on Data Engineering (ICDE)*. 668–681.

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

[61] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *Proceedings of International Conference on Software Engineering (ICSE)*. 225–236.

[62] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 329–339.

[63] David Patrick Reed. 1978. *Naming and Synchronization in a Decentralized Computer System.* Technical Report.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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 International Conference on Software Engineering (ICSE)*. 2072–2084.

[69] Xiaohui Song and Jane W-S Liu. 1990. Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency. In *Proceedings of Annual International Computer Software and Applications Conference (COMPSAC)*. 132–139.

[70] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 294–305.

[71] 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.

[72] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 520–531.

[73] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *Proceedings of International Conference on Software Engineering (ICSE SEIP)*. 328–337.

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

[75] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 1629–1642.

[76] 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)*.

[77] 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 International Symposium on Software Testing and Analysis (ISSTA)*. 302–313.

[78] 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)*. 955–970.

A STUDY METHODOLOGY

Due to the space limit, we cannot present the detailed process about our study methodology in Section 3. In this appendix, we provide some supplementary details about the TXBug collection and categorization process.

A.1 Collecting TXBugs

In our empirical study, we adopt the following process to identify relevant DBMSs and TXBugs.

1) To make our studied TXBugs representative, we select six well-developed and popular DBMSs as our target systems, i.e., MySQL [11], PostgreSQL [14], SQLite [15], MariaDB [9], CockroachDB [5], and TiDB [19]. These systems cover various types of DBMSs (i.e., traditional, embedded and distributed DBMSs), and a diverse set of transaction features (e.g., transaction types, isolation levels and concurrency control mechanisms). Among them, MySQL, PostgreSQL, SQLite and MariaDB are well-developed DBMSs and rank high in the DB-Engines Ranking (2nd, 4th, 10th and 13th, respectively). CockroachDB and TiDB are two popular distributed DBMSs in the open source community (ranked as 3rd and 5th among DBMSs in GitHub).
2) We manually collect issues that are potentially related to TXBugs from the above six DBMSs' issue repositories.
    a) Our studied DBMSs usually evolve quickly. To keep our study results up-to-date, we only consider issues confirmed by DBMS developers in the last 5 years, i.e., from January 2018 to December 2022.
    b) Our target DBMSs contain a huge amount of issues, e.g., MySQL contains about 20,000 issues in the last 5 years. In these DBMSs, they usually do not label whether an issue is related to transaction bugs. It is also challenging and time-consuming to manually check all the reported issues. Therefore, we use the following keywords, i.e., "transaction", "abort", "commit", "isolation level", and their variations, to effectively collect potentially relevant issues in these DBMSs. This search returns us with 7,775 issues.
3) Note that the above search keywords are general and widely used in DBMSs. Although some issues contain our search keywords, they are unrelated to transaction processing mechanisms in fact. To exclude these issues that are not related to transaction processing mechanisms from our study, we manually inspect the bug description and developer comments for each retrieved issue, and check whether the issue is related to transaction processing mechanisms in DBMSs. We only keep the issues that are related to transaction processing mechanisms. In this step, we filter out 7,220 issues, and keep 555 issues for further investigation.
4) For each remaining issue, we carefully investigate its bug description, test cases, developer discussions and available fixing patches, and further rebuild its bug test case step by step, and use the following criteria to select TXBugs.
    a) We keep an issue as a TXBug if it needs at least one explicit transaction or XA transaction to trigger, or it needs at least two transactions to trigger.
    b) If a bug can be triggered by only one auto transaction, we do not consider it as a TXBug, since it usually belongs to transaction-unrelated bugs, e.g., logic bugs in single SELECT statements.
    c) To maintain the accuracy of our study results, we only keep TXBugs that we can completely understand.

To maintain the accuracy of our study, each issue has been investigated by at least three authors. For each issue, its investigators carefully discuss it and reach a consensus. If we cannot reach a consensus about an issue after discussion, we also exclude it from our study. Finally, we obtain 140 TXBugs in our study.

A.2 Categorizing TXBugs

We adopt the open card sorting approach to build the categories for TXBugs' triggering conditions, root causes, impacts and fixing. The basic categorization process is described as follows.

1) For each TXBug, we extract necessary information from its issue report, e.g., test cases, expected execution results, developer discussions, and available fixing patches, and then write down its test case step by step. All the information is cross-checked by at least three authors.
2) Each TXBug is labeled by at least three investigators (i.e., authors) independently. Each investigator tries to classify a TXBug into an existing category in a dimension (e.g., root cause and bug impact) according to the extracted information. If a bug cannot be classified into any existing category, the investigator creates a new category for it.
3) For each TXBug, its investigators discuss their categorization results and try to reach a consensus. If some investigators have disagreements about a TXBug's categorization result, they reinvestigate it further until they reach a consensus. They also refine categories in this step if necessary.
4) To ensure correctness and consistency, step 3 and 4 are performed multiple rounds, until all investigators reach a consensus for all transaction bugs' categorization results.

Specifically, we apply the following process to categorize the studied transaction bugs.

- **Triggering conditions.** Triggering conditions are usually hidden in a TXBug's test case, e.g., the number of transactions and the types of the involved transactions. Each investigator extracts these triggering information from a bug's test case, and classifies TXBugs according to these triggering information.
- **Root causes.** For root causes, we try to answer the question "what transaction semantics does a transaction bug violate?". We investigate root causes from a bug's transaction test case, expected execution results and developers' discussions. Initially, we create four categories related to ACID properties (i.e., atomicity violations, consistency violations, isolation violations and durability violations). During our investigation, we further reveal more categories, e.g., read-only constraint violations and statement correctness violations. These are surprising categories to us. We do not find any TXBugs for durability violations, so we delete this category in our study.
- **Impacts.** For bug impacts, we try to answer the question "what consequences does a transaction bug cause?". We investigate bug impacts from a bug's transaction test case and developers' discussions about the unexpected execution results of the test case. We also reproduce some TXBugs if possible, and observe their consequences.
- **Fixing.** We extract necessary information from issue reports, and available fixing patches. Then, we classify TXBugs according to these information.