ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exploring the Correlation between Transactional Bugs and Isolation Levels

Master Thesis

Theodor Moroianu

December 3, 2024

Advisors: Prof. Dr. David Basin, Dr. Si Liu
Department of Computer Science, ETH Zürich

Abstract

This thesis aims to create a testbed for easily replicating and analyzing Database Management System (DBMS) transactional or isolation bugs, use this testbed to replicate and analyze a set of known bugs, and develop a novel bug-finding tool.

First, we create a testbed leveraging containerization technology to easily spin-up and run a variety of custom versions of DBMSs. The testbed provides an easy way of running concurrent transaction workloads, and generates logs of the test cases being executed. It also provides an easy way of starting multiple MySQL shells connected to arbitrary versions of the *MySQL*, *MariaDB* and *TiDB* DBMSs.

We then use the testbed to replicate and analyze a set of known bugs in *MySQL*, *MariaDB* and *TiDB*. We find that a majority of the replicated bugs happen on all isolation level supported by the DBMS, and we then analyze in depth the bugs manifesting under only a subset of the isolation levels, which can happen either because the bug does not affect some isolation levels, or the proof-of-concept (PoC) test cases included in the bug report do not trigger the bug.

Finally, we discuss subsequent steps and focus on the *TxCheck* fuzzer, developed at ETH Zürich. Building upon the theoretical foundation of its design, which frames database misbehavior as graph constructions, we discover an innovative method to improve the technique. *TxCheck* relies on transaction dependency graphs computed through comprehensive but unsound SQL-level instrumentation. We provide the necessary constructs to achieve a sound graph extraction technique. Additionally, we implement this technique within the *TxCheck* fuzzer.

Acknowledgement

I am very grateful for the opportunity of working on my Master's thesis as part of the Information Security group. I would like to extend my deepest gratitude to Prof. Dr. David Basin for the opportunity to work as part of his group, and to Dr. Si Liu for the help, flexibility, guidance and support he offered me throughout the project.

I am also grateful to my friends Constantin and Alex, my girlfriend Emma and my family for their camaraderie and support throughout the project. Special thanks go to Lucian Bicsi, who proof-read this report.

Lastly, I would like to acknowledge the financial support provided by the ETH Foundation, as part of my ESOP scholarship, which made my studies at ETH possible.

Contents

Chapter 1

Introduction

1.1 Problems and Motivations

Modern database management systems, which often rely on a relational model, were introduced in the 1970s [1]. Since then, the amount of data that needs to be stored and processed and the use cases of DBMSs has grown exponentially, which lead to the development of an entire industry of database software. The growing discrepancy between storage capacity and processing power, and the cost efficiency of buying multiple smaller machines [2] pushed towards the development of concurrency mechanisms and of distributed databases, able to distribute load across multiple machines and users. Distributed databases are essential for virtually all modern large-scale applications, such as social networks, e-commerce, cloud computing or research.

Like all software, DBMSs are prone to bugs, especially considering the diminishing returns of optimizing their performance, which is in many cases the bottleneck of the entire system. While unit and integration tests are essential in the development of any software [3], they are not enough to ensure the correctness of such complex systems. This is why, many current testing strategies rely on the use of fuzzing, a technique that generates random inputs to the system under test, in order to find bugs [4].

The motivation and goal of this project is to replicate existing DBMS transactional bugs reported to their respective issue trackers, and reported by or analyzed in other works [5, 6, 7, 8], in order to understand how they correlate with isolation levels. Then, using the gained insights and using a novel fuzzing technique introduced by Jiang, Z. et al. [5] based on SQL instrumentation, we try to find new bugs. We aim to generate the set of Adya dependencies [9] of randomly generated concurrent transactions, and to use this information to detect bugs in the concurrency and isolation mechanisms of a distributed database.

1.2 Contributions

Overall, we make the following contributions in this project.

1. We develop a new bug testing, replication and analysis framework, leveraging containerization techniques, for starting specific versions of DBMS servers, and automatically replicating DBMS bugs.
2. Using the testing framework, we replicate 135 bugs occuring in the *MySQL*, *MariaDB* and *TiDB* DBMSs.
3. We analyze the reports of the replicated bugs, and we explore the corelation between isolation levels and the reported bugs. We find that a majority of the replicated bugs happen on all isolation levels supported by the DBMS, and we then analyze in depth the bugs manifesting under only a subset of the isolation levels.
4. As follow-up research, we build on a black-box fuzzing technique developped at ETH Zürich for the *TxCheck* fuzzer, by improving its transaction dependency extraction, based on SQL instrumentation and Adya dependency graphs.
5. We then implement our enhancements to the fuzzing technique into *TxCheck*, improving its ability to detect transactional anomalies by integrating SQL instrumentation and Adya dependency graph analysis into the existing framework [5].

Chapter 2

Background

2.1 Database Management Systems

Modern database management systems (DBMS) are complex software systems that provide a high-level interface for users to interact with the underlying data. DBMSs such as *MySQL* [10] offer a large set of features, including data storage, retrieval and manipulation.

Relational DBMSs, usually exposing *SQL* as a query language, form an overwhelming majority of the database systems in use today, with the 4 most popular DBMSs being relational [11]. The relational model was introduced by Edgar Codd in 1970 [1], and offers application developers a high-level manipulation capacity of the stored data. Information is modeled as collections of relations between properties, commonly represented as tables and rows.

Modern *SQL* offers a *Data Definition Language* (DDL) to create and modify the structure of the underlying data, and a *Data Manipulation Language* (DML) to interact with the data. The DDL is composed of statements such as *CREATE* and *DROP*, while DML is composed of statements such as *SELECT*, *INSERT*, *UPDATE*, and *DELETE*.

2.2 Transactions

A transaction is a sequence of instructions executed as a single isolated unit of work. In other words, either all of the instructions in the transaction are correctly executed and saved to the database, or none of them are. Transactions offer the ACID properties [12], a set of properties that guarantee that database transactions are processed reliably. The ACID properties are as follows:

- **Atomicity**: A transaction is an atomic unit of work, meaning that the database will either execute all of the instructions in the transaction, or

none of them.

- **Consistency**: A transaction will bring the database from one consistent state to another consistent state. In other words, the database will always be in a consistent state, regardless of the state of the running transactions.

- **Isolation**: Multiple transactions can be executed concurrently, and, depending on the isolation level, the transactions will not interfere with each other.

- **Durability**: Once a transaction is committed successfully, the DBMS guarantees that the changes made by the transaction will be saved to the database.

2.3 Isolation Levels

The isolation between transactions is defined by the isolation level. Stricter isolation levels offer more consistency guarantees, at the cost of concurrency and performance. While many isolation levels have been formalized, the ANSI isolation levels supported by most database systems [13] are as follows:

- **Read Uncommitted**: The lowest isolation level. Transactions can read uncommitted data from other transactions.

- **Read Committed**: Transactions are visible only after being committed.

- **Repeatable Read**: Transactions are visible only after being committed, and multiple reads of the same data will return the same result. Note that new data can be become visible to the transaction.

- **Serializable**: The strongest (and slowest) isolation level, in which transactions can be assumed to be executed serially.

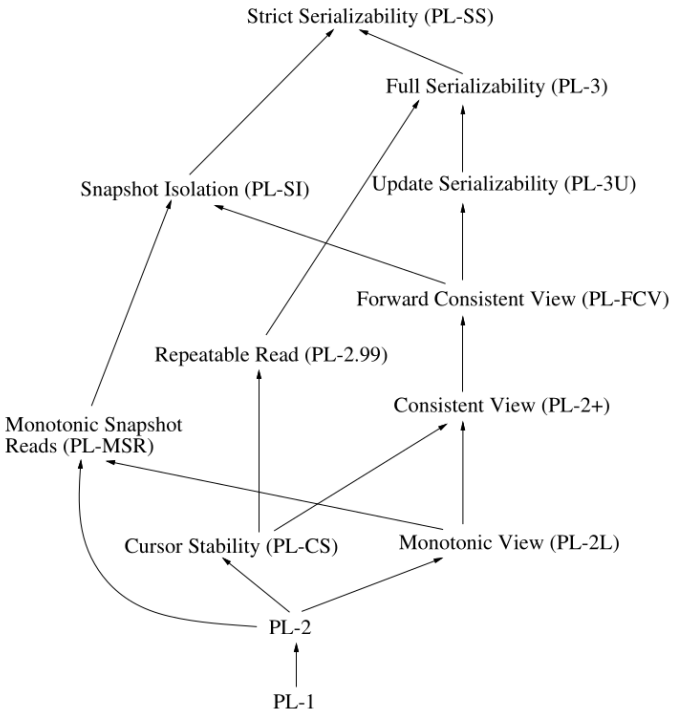

Figure 2.1: The Adya isolation levels [9].

In modern DBMSs, the isolation levels are implemented by leveraging concurrency control techniques such as locking and multi-version concurrency control (MVCC). The choice of isolation level is a trade-off between consistency guarantees and concurrency performance, and is usually made by the application developer.

For instance, a banking application which needs to avoid double-spending will use the *Serializable* isolation level, while a school grading system might want to use the *Read Committed* isolation level.

2.4 Transaction and Isolation Bugs

DBMSs are complex software systems, with their complexity constantly increasing as diminishing returns push for more and more complex optimizations. Like all software, DBMSs are prone to bugs, which can lead to data corruption, loss of data, or crashes.

Transaction and isolation bugs are part of a specific class of bugs residing in the transaction and isolation handling mechanisms of a DBMS. Such bugs are tricky to detect, as they often require multiple concurrent transactions, might

occur sporadically due to the nondeterministic nature of the concurrency, and might not be easily reproducible.

While similar, transactional and isolation bugs are slightly different:

- **Transactional bugs**: Logic bugs that occur when one or multiple transactions are being run. Possible manifestations include unexpected failures, missing data, or incorrect behavior.
- **Isolation bugs**: Bugs that occur when the specified isolation level is not respected. Possible manifestations include forbidden behavior, such as dirty reads, non-repeatable reads, or phantom reads.

2.5 Related Work

The topic of database testing is not new, but has gained significant interest over the last few years, with multiple new techniques and tools developed to ensure the reliability and correctness of database management systems. At the core of this research field lies the need to detect, replicate and understand logical and transactional bugs (which affect the core functionality of the DBMS system, respectively which violate constrainsts imposed by the isolation level).

Recent work has focused on fuzzing techniques, combined with novel methods of detecting errors in random transactions [5, 6, 7, 14]. A recent paper by Cui, Z. et al. [8] makes a comprehensive survey of reported transactional bugs, a large portion discovered with the help of the before-mentioned fuzzing techniques.

Cui et al. proposed *DT2*, a testing framework based on differential testing, a technique that compares the output of the same statements run on two different DBMSs [6]. By generating random database states and executign concurrent transactions, *DT2* was able to identify 10 new bugs across *MySQL*, *MariaDB* and *TiDB*. The choice of the DBMS systems picked for testing is not accidental, as they are some of the most popular DBMSs in use today, and *MariaDB* and *TiDB* are advertized as drop-in replacements for *MySQL*.

From a different approach, Clark, J. et al. [14] proposed *Emme*, a white-box testing framework for detecting isolation bugs by checking violations of the Adya dependency graph [9]. The bug-finding technique relies on existential constraints imposed on bug-valid traces of DBMS transactions, expressed as constraints on a directed graph. The tool relies on *version certificate recovery*, a technique that injects versioning on database records, and *version certificate validation*, a technique which extracts the Adya dependency graph from the versioned records. While the authors of the paper did not find any new bugs, they were able to validate the correctness of their approach by detecting manually injected bugs.

Similarely, Jiang, Z. et al. [5] proposed a novel technique for finding isolation bugs using SQL instrumentation. The technique relies on a similar approach to *Emme*, but the idea of instrumenting SQL queries with additional checks allows the running the instrumented queries in a black-box fashion. The authors are unable to generate a sound Adya dependency graph, but are able to generate single transactions histories equivalent to concurrent ones, allowing for a differential-testing based approach. The authors were able to find 56 new bugs in *MySQL*, *MariaDB* and *TiDB*.

Our work is inspired by the comprehensive surveying work of Cui, Z. et al. [8] on transactional bugs. The survey cataloged a wide range bugs across multiple DBMSs and reported from various fuzzing tools or simple database users, and captures recurring patterns. However, as the authors acknowledge, their analysis of bugs is "*based on their issue descriptions, embedded test cases, developer discussions, and available fixing patches*", and, to the best of our knowledge, the authors of the survey did not actually replicate the collected bugs, due to constraints on the DBMS versions (often a Git commit) and time consumption.

Our work aims to build on the work of Cui, Z. et al. [8] by replicating a subset of the transactional bugs reported in the survey, and by extending the work to isolation bugs. We aim to replicate the bugs in a controlled, container-based environment, and to understand the root cause of the bugs by analyzing the transaction histories.

In the second part of our project, we also build on the work of Clark, J. et al. [14] which find bugs by checking violations of the Adya dependency graph [9] in a white-box fashion, and on the work of Jiang, Z. et al. [5] which introduces the novel idea of SQL instrumentation. Using these two techniques, we introduce a new technique for finding isolation bugs using Adya dependency graphs in a black-box fashion, leveraging the SQL instrumentation technique.

Chapter 3

Developing a DBMS Transactional Testing Framework

3.1 Overview

This chapter presents the design, implementation and usage of a testing framework for replicating DBMS transactional bugs. Using the testing framework, we replicate a set of transactional bugs in the *MySQL*, *MariaDB* and *TiDB* DBMSs. We then analyze the reports of the replicated bugs, and we explore the corelation between isolation levels and the reported bugs.

3.2 Design

The testing framework, is implemented in *Python*, and heavily relies on *Podman*, a container manager [15] for managing DBMS instances. The tool works on `x64 GNU/Linux` systems, and we developed it in *VSCode*, with the help of *Github Copilot* [16].

The framework is modular, helping any future developer to easily extend it (for instance for adding support for new DBMSs). The main components in the bug testing pipeline (see Figure 3.1) are the following:

- The *podman connector*: This component handles the interaction with the *Podman* engine, and is responsible for starting, stopping, downloading and managing containers running DBMS instances.
- The *test parser*: This component handles the parsing of test cases, using a specific format, and is responsible for creating the internal representation of the test cases.
- The *mysql connector*: This component handles the connection to a DBMS instance (running within a container), and is responsible for executing statements in order and extracting the results.

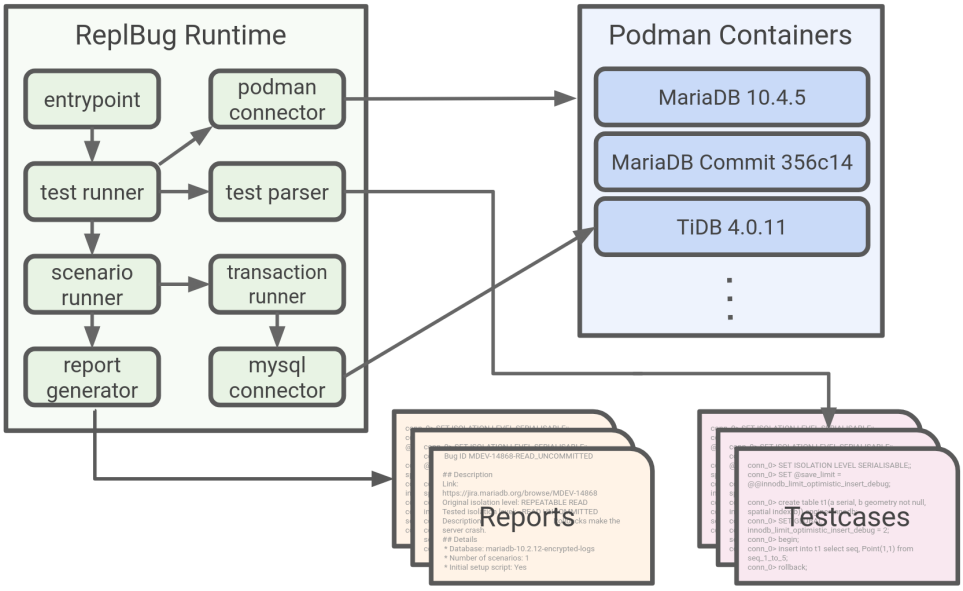

Figure 3.1: Design of the *ReplBug* testing framework

- The *transaction runner*: This component handles the execution of all the statements in a transaction, and runs on different threads for concurrency.
- The *scenario runner*: This component runs test cases under a specific configuration.
- The *test runner*: This component orchestrates the execution of all required test cases under all specified configurations.

3.3 Custom DBMS Version

Some bugs are specific to a certain version of a DBMS, which might not be available as pre-built binaries. For instance, versions with serious vulnerabilities are usually removed from official repositories, or intermediary versions tied to a specific *Git* commit are not released as binaries.

We thus consider the ability to built DBMSs from source (given a version tag or a *Git* commit) essential. To simplify the process, we provide sample *Dockerfile* templates, which can be used to test specific DMBS versions. A sample *Dockerfile* for *TiDB* can be seen in Figure 3.2.

In our project, we provide *Dockerfiles* for *MySQL*, *MariaDB* in release or debug mode, and *TiDB* with or without *TiKV*. Creating a new docker file only requires the *Git* commit, and then running the *build* command integrated into *ReplBug*.

```
FROM golang:1.19-alpine AS builder

# Install git and other dependencies
RUN apk add --no-cache git make bash gcc wget binutils-gold \
    musl-dev curl tar

# Set the working directory inside the container and
# create necessary directories
RUN mkdir -p /go/src/github.com/pingcap
WORKDIR /go/src/github.com/pingcap

ARG TIDB_COMMIT=c9288d246c99073ff04304363dc7234d9caa5090

# Clone and build the TiDB repository
RUN git clone --depth 1 https://github.com/pingcap/tidb.git \
    && cd tidb \
    && git fetch --depth 1 origin "$TIDB_COMMIT" \
    && git checkout "$TIDB_COMMIT" \
    && make -j \
    && mv bin/tidb-server /usr/local/bin/tidb-server \
    && cd .. \
    && rm -rf tidb

EXPOSE 4000
WORKDIR /usr/local/bin
CMD ["./tidb-server", "-P", "4000"]
```

Figure 3.2: Sample *Dockerfile* for building a specific version of *TiDB*

3.4 Testing Meta-Language

For a given test case, specifying the statements and their execution order on the DBMS is hard, due to multiple reasons:

- Transactional and isolation bug PoCs usually need multiple concurrent transactions, making a simple *SQL* script insufficient.
- Some statements are expected to fail, which might lead to the termination of a standard script.
- The order of the statement execution (and sometimes the locking order) is important for the bug to manifest.

To address these issues, the *MySQL* development team created a testing framework which encodes test cases in a special format [17]. Using this

```
ORIGINAL_ISOLATION_LEVEL = DEFAULT_ISOLATION_LEVEL
BUG_ID = "MDEV-26642"
LINK = "https://jira.mariadb.org/browse/MDEV-26642"
DB_AND_VERSION = db_config.DatabaseTypeAndVersion(
    db_config.DatabaseType.MARIADB, "10.6.17"
)
SETUP_SQL_SCRIPT = """
create table t(a int, b int);
insert into t values (0, 0), (1, 1), (2, 2);
"""

DESCRIPTION = "The last select does not respect the update
               (a should always be 10)."


def get_scenarios(isolation_level: IsolationLevel):
    return [
        f"""
        conn_0> SET GLOBAL TRANSACTION ISOLATION LEVEL
                                {isolation_level.value};
        conn_0> begin;
        conn_0> select * from t;
        conn_1> begin;
        conn_1> update t set a = 10 where b = 1;
        conn_1> commit;
        conn_0> select * from t;
        conn_0> update t set a = 10 where true;
        conn_0> select * from t;
        conn_0> commit;
        """,
    ]
```

Figure 3.3: Replication script for the bug *MDEV-26642* in *MariaDB 10.6.17*.

format, however, is cumbersome, as we only need a small subset of the features, and using the *MySQL* interpreter would make it hard to test other DBMSs.

We thus create a small, custom scripting language on top of *Python*, inspired by the way bug reporters describe their PoCs. In Figure 3.3, we present a sample test case for the bug *MDEV-26642* in *MariaDB 10.6.17*.

Each test case provides the following information:

- The *DBMS* and version on which the bug was reported.

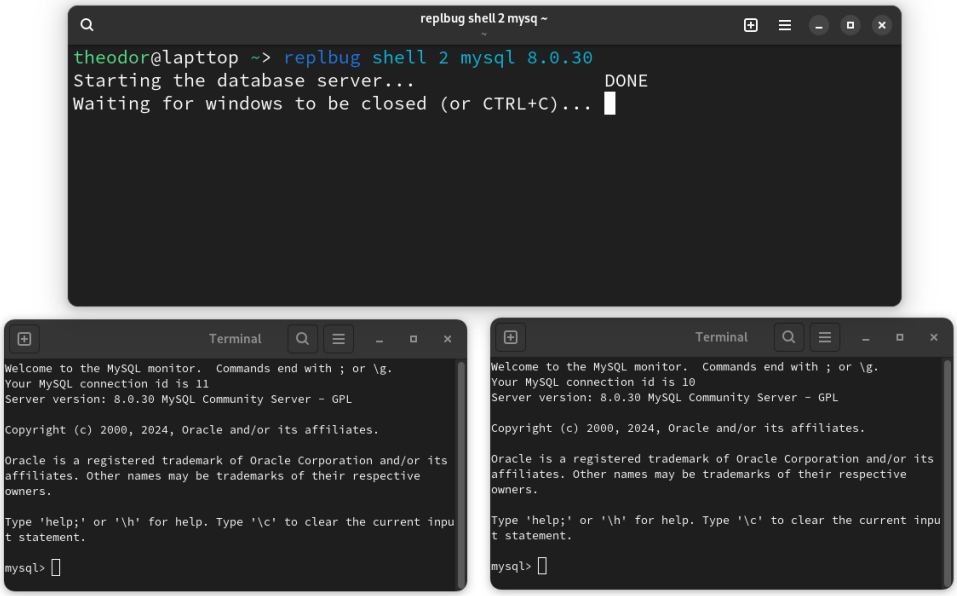

Figure 3.4: Using *ReplBug* to start 2 *MySQL v8.0.30* shells

- The bug ID and a link to the bug report.
- The setup script, which is executed before the test cases. If the setup script is too long, it can be stored in a separate file.
- The description of the bug.
- The scenarios, which are the test cases that will be executed (one for each isolation level). Each scenario is a sequence of statements, executed concurrently by different transactions.

For running a test case, the tool provisions the required DBMS instance, executes the setup script (if present), and then runs the scenarios under all supported isolation levels. For each transaction a separate connection to the DBMS server is created. The results are then stored in a report, which can be further analyzed.

3.5 Usage

The testing framework, called *ReplBug* is invoked from the CLI. The main features it offers, exposed by the executable as subcommands are the following:

- `shell` (See Figure 3.4): Starts one or multiple *MySQL*, *MariaDB* or *TiDB* shells, connected to a specific version of the DBMS. If the version is not present on the local machine, the tool will attempt to pull the image from Docker Hub.

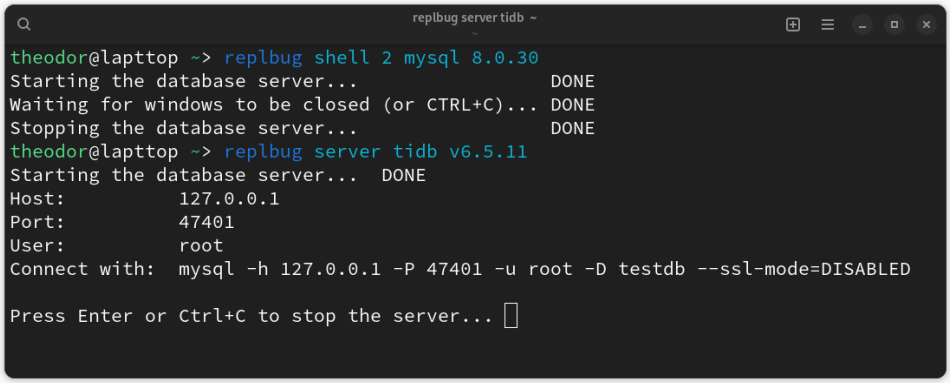

Figure 3.5: Using *ReplBug* to start a *TiDB v6.5.11* server

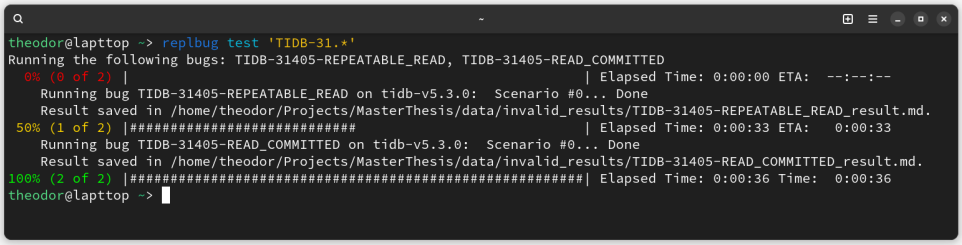

Figure 3.6: Using *ReplBug* to generate reports of some known bugs

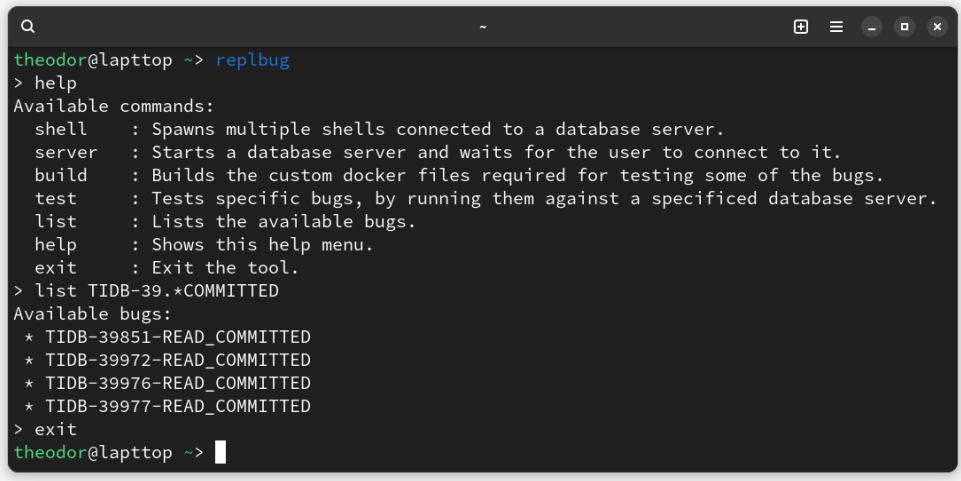

Figure 3.7: Using *ReplBug* in interactive mode

- `server` (See Figure 3.5): Starts a specific version of the *MySQL*, *MariaDB* or *TiDB* DBMS and provides the required details (host, port, user) for connecting to the server.
- `test` (See Figure 3.6): Runs the scenarios of some known bugs (which have to be written in a specific format prior), and automatically generates reports of the execution.
- `list`: Returns a list of the test cases available in the tool (optionally a *regex* can be passed to filter the results).

The tool can be either used from the CLI by passing arguments, or in interactive mode, where the tool exposes a shell that can be used by the user (see Figure 3.7).

Chapter 4

Replicating Transactional Bugs in MySQL, MariaDB and TiDB

4.1 Overview

With the help of the *ReplBug* testing framework, we were able to replicate *MySQL*, *MariaDB* and *TiDB* transactional, logical and isolation bugs. We focus on transaction and isolation bugs, reported by DBMS testing papers [8, 7, 6]. We then replicate the bugs on the same versions of the DBMSs, and verify which isolation levels are affected. The number of bugs taken from each paper can be seen in Figure 4.1.

4.2 Replicated Bugs

We try to replicate *MySQL* and *MariaDB* bugs on the 4 isolation levels supported by the DBMSs (*Read Uncommitted*, *Read Committed*, *Repeatable Read* and *Serializable*). For *TiDB*, we replicate the bugs on the 2 isolation levels supported by the DBMS (*Read Committed* and *Serializable*).

The methodology for replicating a bug is the following:

- We search for the list of bugs reported by a paper, and filter the bugs that are transactional or isolation bugs (as reported by the paper), and manifest on one of the DBMSs we support (*MySQL*, *MariaDB* and *TiDB*).
- We read the bug report in the DBMS's bug tracker, extract the version of the DBMS on which the bug was reported and a PoC.
- If the DBMS version is not available as a pre-built binary, we build the DBMS from source using the *Dockerfile* templates provided by the *ReplBug* tool.

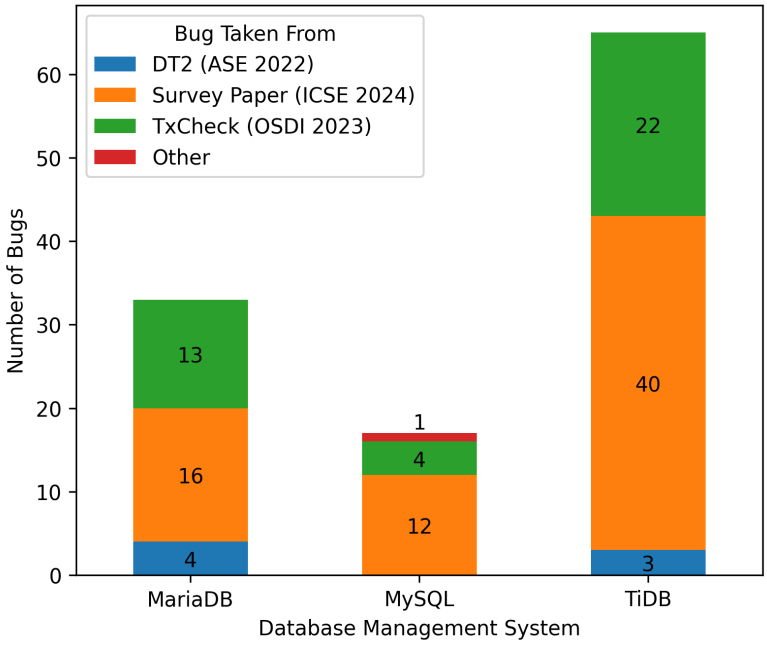

Figure 4.1: Distribution of bugs by DBMS and reporting paper [8, 7, 6].

- We write a test case in the *ReplBug* testing framework, using the meta-language described in the previous chapter.
- We run the test case on the DBMS, under all supported isolation levels.
- We sometimes have to adjust the test case, as some reports do not include the exact version of the DBMS, or the PoC is precise enough.

Within this project, we successfully replicated 115 bugs, out of which 33 manifest on the *MariaDB* DBMS, 17 manifest on *MySQL* and 65 manifest on *TiDB*. The distribution of the bugs by DBMS and reporting paper can be seen in Figure 4.2.

4.3 Analysis

We run the *ReplBug* testing framework on the selected bugs, and we generate reports of their execution. We then read the reports, and analyze the output of the test cases. We then explore the corelation between the bugs and the isolation levels.

We find that 100 bugs manifest on all isolation levels supported by the DBMS, and 15 bugs only manifest on a subset of the isolation levels. In the reminder of this chapter, we refer to the bugs that manifest only on a subset of the

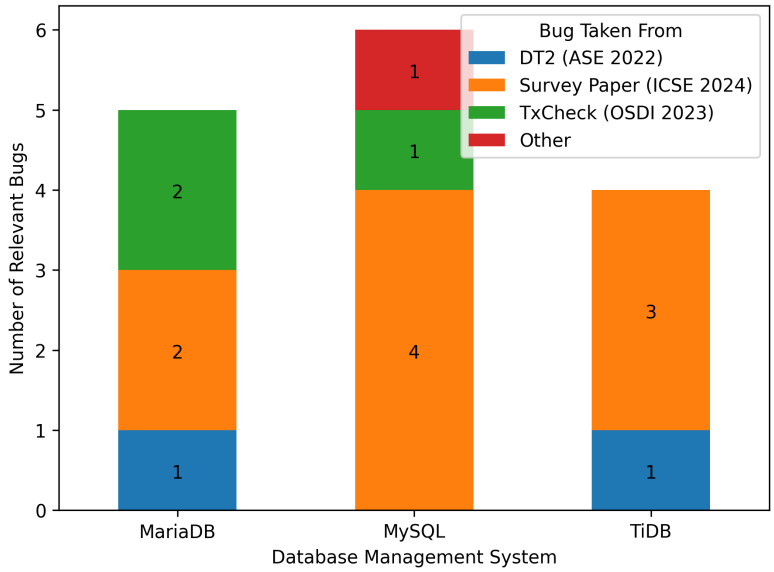

Figure 4.2: Relevant bugs by DBMS and reporting paper [8, 7, 6].

isolation levels as *interesting bugs*. In Figure 4.2, we present the distribution of the *interesting bugs* by DBMS and reporting paper.

We then explore the isolation levels under which the *interesting bugs*, manifest. We find that:

- 5 bugs manifest under *Read Committed* and *Repeatable Read*.
- 4 bugs manifest under *Repeatable Read*.
- 2 bugs manifest under *Read Uncommitted* and *Read Committed*.
- 2 bugs manifest under *Read Committed*.
- One bug manifests under *Serializable*.
- One bug manifests under *Repeatable Read* and *Serializable*.

The findings are illustrated in Figure 4.3. The main limitation of this analysis is that the number of *interesting bugs* is small, making the results not statistically significant. Additionally, our approach only allows us to verify if a bug is triggered by a specific PoC, and not to explore the root cause of the bug, which could in theory be triggered by other PoCs under different isolation levels.

4.4 Interesting Bugs

We present a brief overview of the *interesting bugs*, and provide a plausible explanation for their behavior, where possible. The bugs are presented in the

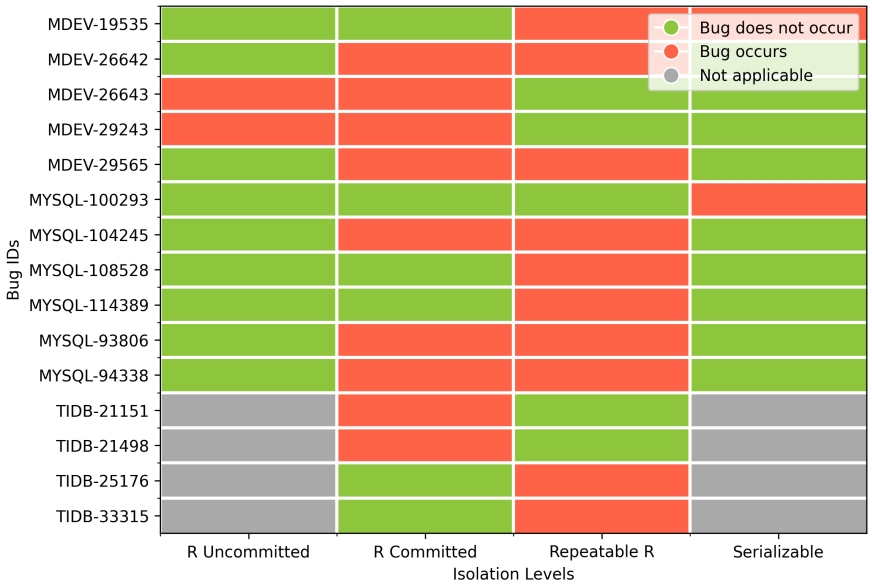

Figure 4.3: Relevant bugs by isolation levels.

order of the number of isolation levels under which they manifest.

Bug MDEV-19535

This bug is replicated on *MariaDB 10.4.5*, and manifests under *Repeatable Read* and Serializable.

For compatibility, *MariaDB* provides an `sql_mode` variable, which can be used to mimic the behavior of other DBMSs.

When the `sql_mode` is set to `ORACLE`, *MariaDB* omits to add exclusive locks when running an *SELECT FOR UPDATE* statement. This leads to incorrect behavior when running *SELECT FOR UPDATE* statements under *Repeatable Read* and *Serializable* isolation levels, as reads are no longer guaranteed to be repeatable.

Bug MDEV-26642

This bug is replicated on *MariaDB 10.6.17*, and manifests under *Read Committed* and *Repeatable Read*. The bug was fixed by a PR in version 10.6.18, and was marked as affecting *MySQL* too.

The bug affects concurrent modifications of the same table: if a transaction updates a row of the table, and another transaction updates the entire table, the second transaction does not see its own modifications. The main part of the PoC can be seen in Figure 4.4.

```
conn_0> begin;
conn_0> select * from t;        -- [(0, 0), (1, 1), (2, 2)]

conn_1> begin;
conn_1> update t set a = 10 where b = 1;
conn_1> commit;

conn_0> select * from t;        -- [(0, 0), (1, 1), (2, 2)]
conn_0> update t set a = 10 where true;
conn_0> select * from t;        -- [(10, 0), (1, 1), (10, 2)]
conn_0> commit;
```

Figure 4.4: PoC for the bug *MDEV-26642*.

```
conn_0> insert into t values(null, 1), (2, 2),
                    (null, null), (null, 3), (4, null);
conn_0> begin;
conn_0> update t set a = 10 where 1;
conn_1> begin;
conn_1> update t set b = 20 where a;
conn_0> commit;
conn_1> commit;
conn_2> select * from t;
        -- [(10, 1), (10, 20), (10, 20), (10, 20), (10, 20)]
```

Figure 4.5: PoC for the bug *MDEV-26642*.

The bug is caused by a design issue of *InnoDB*, the storage engine used by both *MariaDB* and *MySQL*.

The bug is due to the inability of *InnoDB* to detect *write-write* conflicts. Under *Read Committed* and *Repeatable Read*, *InnoDB* creates a read view at the start of a statement / transaction, used for knowing which records should be visible, but does not handle overwriten records properly.

The *Read Uncommitted* isolation level always displays the latest state of the table, and *Serializable* will lock the accessed records first, so the bug does not manifest under these isolation levels.

Bug MDEV-26643

This bug is replicated on *MariaDB 10.5.12*, and manifests under *Read Uncommitted* and *Read Committed*. The bug was fixed by a PR. The PoC is very similar to *MDEV-26642*, and can be seen in Figure 4.5.

```
conn_1> START TRANSACTION;
conn_1> update t set a = 162;
conn_0> START TRANSACTION;
conn_1> COMMIT;
conn_0> select * from t where <CONDITION>; -- returns 1 record
conn_0> update t set a = 63 where <CONDITION>;
conn_0> select * from t where a = 63;      -- returns 2 records
conn_0> COMMIT;
```

Figure 4.6: Simplification of the PoC for the bug *MDEV-29565*.

The bug is caused by an improperly used semi-consistent read, where *Read Uncommitted* and *Read Committed* transactions are sometimes not updated with the latest changes.

Bug MDEV-29243

This bug is replicated on *MariaDB 10.8.3*, and manifests under *Read Uncommitted* and *Read Committed*, by causing a crash of the DBMS server.

The root cause of the bug is an incorrect and redundant check of the data retrieval status in *InnoDB*, which leads to potential assertion failures. The bug was fixed by erasing the redundant check.

Bug MDEV-29565

This bug is replicated on *MariaDB 10.8.3*, and manifests under *Read Committed* and *Repeatable Read*.

While confirmed as intended behavior, this is caused by an poorly documented feature of *InnoDB*, which allows changes made by other transactions to be visible in the current one if changes are made to the same records [18]. The PoC is simplified in Figure 4.6.

Bug MYSQL-100293

This bug is replicated on *MySQL 5.7.31*, and manifests under *Serializable*.

When the `innobase_query_caching_of_table_permitted` flag is set to `true` (by passing `--query-cache-type=1` as an argument to the server), *Serializable* transactions are not blocked when using the cache, which causes some missing locks.

The bug was fixed by properly handling `--query-cache-type=1` when the transaction isolation level is *Serializable*.

Bug MYSQL-104245

This bug is replicated on *MySQL 8.0.23*, and manifests under *Read Committed* and *Repeatable Read*.

When inserting rows with the same primary key multiple times (using the `INSERT IGNORE` statement), row locks are duplicated. Using `REPLACE INTO` is even worse, and adds many row locks (we believe the number of added locks is the number of matched records times the number of inserted records).

The bug only manifests on *Read Committed* and *Repeatable Read*, as *Serializable* has a different locking mechanism, and *Read Uncommitted* does not lock the records at all.

This bug dramatically increases latency, but does not cause deadlocks or crashes, as all the redundant row locks are identical. The bug is not present in *MySQL 8.0* or later, so it was not fixed.

Bug MYSQL-108528

This bug is replicated on *MySQL 5.7.34*, and manifests under *Repeatable Read*.

The bug is marked as verified, but we strongly consider the PoC actually works as intended (it is not a bug). The PoC is simplified in Figure 4.7.

In the PoC, one transaction updates a table and commits. Another concurrent transaction does not see the changes made by the first transaction, however those changes become visible when the second transaction tried to update a table. This behavior is similar to the behavior of *MDEV-29565*, and we believe it is due to the same poorly-documented feature of *MySQL* [18].

Bug MYSQL-114389

This bug is replicated on *MySQL 8.0.12*, and manifests under *Repeatable Read*. The bug is still present in the latest version of *MySQL* (version *9.1.0*). The report was closed as *duplicate*, but the original bug report is still open.

As the bug is not fixed yet, we do not know the root cause. The PoC of the bug can be seen in Figure 4.8.

Bug MYSQL-93806

This bug is replicated on *MySQL 8.0.12*, and manifests under *Read Committed* and *Repeatable Read*. The bug was fixed as part of the *MySQL 8.0.16* release.

The bug is caused by a mishandling of the `INSERT ... ON DUPLICATE KEY` statement. When a record with a conflicting primary key is inserted, the key should be changed and a row lock created. However, under *Read Committed*

```
conn_1> START TRANSACTION;
conn_0> START TRANSACTION;
conn_0> select * from t_rpjlsd;          -- Create snapshot.

conn_1> update t_g6ckkb set wkey = 162; -- Update the table.
conn_1> COMMIT;                          -- Commit.

conn_0> select * from t_rpjlsd where
            t_rpjlsd.c_pfd8ab <= (
                select min(wkey)
                from t_g6ckkb
            );                           -- Affects 1 row.
conn_0> update t_rpjlsd set wkey = 63 where
            t_rpjlsd.c_pfd8ab <= (
                select min(wkey)
                from t_g6ckkb
            );                           -- Affects 2 rows.
```

Figure 4.7: Simplification of the PoC for the bug *MYSQL-108528*.

```
conn_0> BEGIN;

conn_1> BEGIN;
conn_1> UPDATE t SET b = 222, c = 333;   -- Update the table.
conn_1> COMMIT;

conn_2> BEGIN;
conn_2> SELECT pkId, b, c FROM t;        -- Create snapshot.

conn_0> UPDATE t SET a = 40 WHERE a = 44;
conn_0> COMMIT;

conn_2> UPDATE t SET b = 888, c = 999;   -- Update the table.
conn_2> SELECT pkId, b, c FROM t where   -- Should be empty but
          b = 854 or c = 333 order by b; -- returns a row.
```

Figure 4.8: PoC for the bug *MYSQL-114389*.

```
conn_0> create table t(id int primary key, a int)engine=innodb;
conn_0> insert into t values(1,1),(5,5);

conn_0> SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATBLE READ;
conn_0> begin;
conn_0> insert into t values(5,5) ON DUPLICATE
            KEY UPDATE a=a+1; -- Creates a range lock instead
                              -- of a row lock.

conn_1> begin;
conn_1> insert into t values(4, 4); -- Is needlessly blocked by
                                    -- the range lock.
```

Figure 4.9: Simplification of the PoC for the bug *MYSQL-93806*.

and *Repeatable Read*, a range lock is created instead. The PoC is simplified in Figure 4.9.

Bug MYSQL-94338

This bug is replicated on *MySQL 5.7.25*, and manifests under *Read Committed* and *Repeatable Read*. The bug is not present in *MySQL 8.0* or later, so it was not fixed.

The bug manifests by causing dirty-reads when inserting multiple rows on one transaction, and performing a complex query on another transaction. The PoC is simplified in Figure 4.10. We do not know the root cause of the bug, as no bug fix was made available.

Bug TIDB-21151

This bug is replicated on *TiDB v4.0.8* when using the *TiKV* key-value store, and manifests under *Read Committed*. The bug was closed by a PR pushed to the `master` branch on *2020-11-24*.

The bug occurs because the `USE_INDEX_MERGE` feature does not refresh the current timestamp of the transaction, causing the transaction to potentially miss the latest committed writes. While this is correct to *REPETABLE READ* transactions, *READ COMMITTED* transactions are expected to see committed changes. A simplified PoC can be seen in Figure 4.11.

The bug fix correctly updates the timestamp used by transactions when running under *READ COMMITTED*, by invoking the `refreshForUpdateTSForRC` internal function when necessary.

```
conn_0> BEGIN;
conn_1> BEGIN;

conn_1> INSERT INTO t VALUES
        (1,40,'B',10,1),
        (1,41,'B',10,1),
        ...
        (1,40,'C',16,1),
        (1,42,'C',16,1);

conn_0> SELECT * FROM t1
        WHERE <<CON`DITION>>; -- Dirty read.;
```

Figure 4.10: Simplification of the PoC for the bug *MYSQL-94338*.

```
conn_0> BEGIN;

-- Update the table, after transaction 0 started.
conn_1> BEGIN;
conn_1> update t set value = 11 where id = 2;
conn_1> COMMIT;

conn_0> select /*+ NO_INDEX_MERGE() */ *
            from t where a > 3 or b > 3;    -- Ok.

conn_0> select /*+ USE_INDEX_MERGE(t, ia, ib) */ *
            from t where a > 3 or b > 3;    -- Misses the update.
```

Figure 4.11: Simplification of the PoC for the bug *TIDB-21151*.

Bug TIDB-21498

This bug is replicated on *TiDB*, on the custom commit `3a32bd2d` using the *TiKV* key-value store, and manifests under *Read Committed*. The bug was closed by a PR pushed to the `master` branch on *2021-01-12*.

The bug occurs because of missing consistency checks which allow one concurrent transaction to perform DDL (Data Definition Language) operations on a table (like inserting / deleting columns or deleting indexes), while another transaction is reading from the same table. The PoC is simplified in Figure 4.12.

```
conn_0> begin;

conn_1> alter table t drop index iv;
conn_1> update t set v = 11 where id = 1;

conn_0> select * from t where v = 10; -- Returns [1, 10].
conn_0> select * from t where id = 1; -- Returns [1, 11].
```

Figure 4.12: Simplification of the PoC for the bug *TIDB-21498*.

```
conn_0> begin;                                  -- Start txn.

conn_1> update test.ttt set a=2 where id=1; -- Update records.

conn_0> set @@tidb_snapshot=TIMESTAMP(NOW());

conn_0> select a from test.ttt where id=1;            -- [(1)].
conn_0> select a from test.ttt where id=1 for update; -- [(2)].
conn_0> select a from test.ttt where id=1;            -- [(2)].
```

Figure 4.13: Simplification of the PoC for the bug *TIDB-25176*.

Bug TIDB-25176

This bug was reported on a specific commit of the `master` branch, is replicated on *TiDB 4.0.7* when using the *TiKV* key-value store, and manifests under *Repeatable Read*. The bug is still open as of *November 2024*.

The bug is due to the usage of the `tidb_snapshot` system variable, which sets the timestamp delimiting the visibility of the records [19].

Under *Repeatable Read*, setting this timestamp and the performing a `SELECT` statement breaks the isolation level guarantees. The PoC is simplified in Figure 4.13.

Bug TIDB-33315

This bug is replicated on *TiDB 5.4.0* when using the *TiKV* key-value store, and manifests under *Repeatable Read*.

The bug is caused by the mismatch of rows created by concurrent transactions: if a transaction performs a statement matching an entire table while another transaction holds an exclusive lock on the table, the first transaction will see an inconsistent state. The PoC is simplified in Figure 4.14.

```
conn_0> BEGIN PESSIMISTIC;
conn_0> UPDATE t SET c1=2, c2=2;

conn_1> BEGIN PESSIMISTIC;
conn_1> DELETE FROM t;      -- Delete all records.

conn_0> COMMIT;             -- First transaction commits.

conn_1> SELECT * FROM t;    -- Returns one row.
conn_1> COMMIT;
```

Figure 4.14: Simplification of the PoC for the bug *TIDB-33315*.

4.5 Conclusion

In this chapter, we present an analysis of a few isolation bugs we consider *interesting*, as they manifest under a subset of the isolation levels supported by the DBMSs. We provide a brief overview of the bugs, and a plausible explanation for their behavior.

We find that the bugs are caused by a variety of reasons, from design issues of the storage engine to poorly documented features of the DBMSs. We also find that some bugs are quickly fixed, while others are left unfixed until they *fix themselves*, i.e. they are no longer present in the latest versions of the DBMSs.

The main cause of bugs is the improper handling of locking and MVCC (Multi-Version Concurrency Control) mechanisms, which are the core of the transactional guarantees provided by the DBMSs, as *MySQL*, *MariaDB* and *TiDB* define their isolation levels based on locking behavior and transaction visibility [20, 21, 22].

Chapter 5

Detecting Isolation Bugs in DBMSs via Dependency Graphs Construction

5.1 Introduction and Motivation

As discussed in the previous chapter, and in various other works [5, 8, 7, 14, 6], isolation bugs are a prevalent problem in DBMSs, due to the inherent trade-off between performance (under the form of concurrency) and isolation.

Modern database systems usually express isolation guarantees under a *locking behavior*, which is a set of rules dictating how tables and rows are locked during transactions. However, due to the sheer complexity of modern DBMSs, ensuring a correct locking behavior is challenging, due to a combination of factors such as:

- DBMS systems are complex, with multiple components interacting with each other, such as the query planner, the storage engine, the transaction manager, etc. A bug in any of these components can lead to an isolation bug.
- The performance of DBMSs is subject to the law of diminishing returns, requiring more and more complex optimizations to achieve smaller and smaller performance gains. This makes it more likely for bugs to be introduced.
- Higher concurrency leads to better performance, thus making a reduction in the number of locks desirable. However, this also makes it more likely for isolation bugs to occur.

As *isolation bugs* are not necessarily *logic bugs*, i.e. not bugs in the traditional sense but rather a violation of the isolation guarantees, hunting them with standard bug-finding techniques is challenging:

- Testing high concurrency workloads, required for capturing edge cases where isolation bugs occur, is hard to achieve due to an exponential growth in the number of possible states.
- DBMSs are not fully compatible with each others, and concurrent workloads are inherently non-deterministic, making it hard to check for bugs by exploring differences between DBMSs.
- Semi-automatic techniques such as *fuzzing* require an *oracle* to determine if the output is correct, which is hard to achieve in the case of isolation bugs.

Current techniques avoid these problems by using a white-box testing approach [14], using the database system itself as a testing oracle [5], or running workloads on multiple DBMSs to check for disparities [6].

Following our work on the bug replication testbed, we explore potential next steps and decide to focus on *TxCheck*, a database fuzzer, for several reasons:

- *TxCheck* is developed at ETH Zürich, making it a natural choice for in-depth research.
- It has successfully identified an impressive 56 bugs in *MariaDB*, *MySQL*, and *TiDB*, some of which we analyzed using our bug testbed.
- *TxCheck* was released last year (2023), making it highly relevant.

In this chapter, we introduce our work on enhancing a novel bug-finding technique that leverages Direct Serialization Graphs (DSG) introduced by Adya, A. [9] and SQL-level instrumentation introduced by Jiang, Z. [5]. Our approach builds upon and modifies the method used by Jiang, Z. Additionally, we implement a tool based on our technique within the *TxCheck* fuzzer, which we adapt to meet our requirements.

5.2 Database Model Used

For testing the correct behavior of database systems, we need well-defined specifications we can test against. While some techniques rely on the DBMSs themselves for self-validating or cross validating results [6], other bug-finding techniques express DBMS isolation bugs as violated constraints in the Direct Serialization Graph (DSG) built following Adya's data model [5, 14], directed labeled graphs in which nodes are transactions and directed edges are dependencies.

In particular, *TxCheck*, which this project builds on, uses a mixture of both methods:

- Using *SQL instrumentation*, *TxCheck* extracts a super-graph of the DSG. However, the dependency checking technique used is not perfect, and can create sporadic dependencies.
- Using the DSG, *TxCheck* creates and executes an equivalent execution trace using a single transaction on the same DBMS, which is used as an oracle for comparing the results.

In this work, we rely on the Direct Serialization Graph (DSG) data model introduced by Adya, A. [9], and on the SQL-level instrumentation technique introduced by Jiang, Z. [5].

5.3 Contributions

Our two main contributions are:

- We build on the theory behind the DSG extraction used in *TxCheck*, for allowing for a complete and sound extraction of all the DSG dependencies, by leveraging the *SQL instrumentation* technique *TxCheck* introduced. Our approach allows for an accurate extraction of the DSG edges, which avoids the sporadic dependencies created by the original technique, and allows for a more accurate detection of isolation bugs.
- We implement the changes in the *TxCheck* project, giving *TxCheck* the ability to extract the DSG edges accurately.

5.4 Background on DSG Dependencies

Throughout this work, we heavily reuse the definitions and notations introduced by Adya, A. [9]. The most important definitions are the ones regarding the dependencies between transactions, which we outline below.

Definition 5.1 ***Overwriting a Predicate****: T_j overwrites an operation $r_i(P : Vset(P))$ or $w_i(P : Vset(P))$ if:*

- *T_j installs a version x_j for some object x.*
- *The version x_k of x present in $Vset(P)$ is an earlier version than x_j.*
- *x_j matches the predicate P and x_k does not, or vice versa.*

Definition 5.2 ***Directly Item-Read-Depends****: T_j directly item-read-depends on T_i if T_j reads an some object version installed by T_i.*

Definition 5.3 ***Directly Predicate-Read-Depends****: T_j directly predicate-read-depends on T_i if T_j performs a read operation $r_j(P : Vset(P))$ and T_i installs x_i such that $x_i \in Vset(P)$.*

Definition 5.4 ***Directly Read-Depends****: T_i directly read-depends on T_j if T_i directly item-read-depends or directly predicate-read-depends on T_j.*

Definition 5.5 ***Directly Item-Anti-Depends****: T_j directly item-anti-depends on T_i if T_i reads some object version x_k and T_j installs a later version x_j.*

Definition 5.6 ***Directly Predicate-Anti-Depends****: T_j directly predicate-anti-depends on T_i if T_j overwrites the predicate of an operation $r_i(P : Vset(P))$.*

Definition 5.7 ***Directly Anti-Depends****: T_i directly anti-depends on T_j if T_i directly item-anti-depends or directly predicate-anti-depends on T_j.*

Definition 5.8 ***Directly Item-Write-Depends****: T_j directly item-write-depends on T_i if T_i installs a version x_i and T_j installs the next version x_j.*

Definition 5.9 ***Directly Predicate-Write-Depends****: T_j directly predicate-write-depends on T_i if:*

- *T_j overwrites an operation $w_i(P : Vset(P))$, or*
- *T_j executes an operation $w_j(P : Vset(P))$ and T_i installs some version $x_i \in Vset(P)$.*

Definition 5.10 ***Directly Write-Depends****: T_i directly write-depends on T_j if T_i directly item-write-depends or directly predicate-write-depends on T_j.*

The steps for building the DSGs given an ordered sequence of statements (each belonging to a possibly different transaction) are as follows:

1. Run the statements sequentially, saving the versions of the objects at each step.
2. Extract the *Direct Read Dependencies*, *Direct Write Dependencies* and *Direct Anti-Dependencies* between transactions.
3. Build the DSGs using the extracted dependencies.

The second step is the hardest, due to the inherent lack of information a DBMS provides about the internal state of the system. This is where SQL-level instrumentation comes in.

5.5 SQL-level Instrumentation

5.5.1 Intuition Behind SQL-level Instrumentation

The most straight-forward way to extract DSG dependencies is to modify the source code of DBMSs, exposing the internal state of the system. This allows for a relatively simple extraction of dependencies, as the injected code has access to all of the instruction scheduling, existing locks, state of the

key-value store, etc. This is how Clark, J. et al. extract the DSGs in their work [14].

A less intrusive approach, which considers the DBMS as a black box (thus being easily ported across different systems), is to use SQL-level instrumentation, a technique pioneered by Jiang, Z., which consists of systematically exposing the internal state of the DBMS by injecting *SELECT* statements in the SQL queries [5].

The informal intuition behind SQL-level instrumentation is that we want to know the current state of the database, but lack the white-box access to the DBMS internals. As we can only query the database, we infer the dependencies by injecting *SELECT* statements in test cases. We instrument the SQL queries by adding instrumentation code right before and right after each original SQL statement.

A few challenging parts are:

- Ensuring the instrumentation code is not interrupted by locking or other operations. This is addressed by re-ordering statements in the test case, and erasing them if no viable scheduling order can be found.
- Ensuring that every table of the database stores versioning information for each object, required for understanding the view a statement has access to. This is addressed by adding a *version* column to each table.
- Concurrency issues, such as the non-deterministic locking and unlocking behavior of transactions. This is addressed by spacing out the statements temporally, and ensuring that no transaction locking takes place.

5.5.2 Instrumentation Statements

We instrument SQL statements by encapsulating them with instrumentation placed before and after statements, which expose the transaction's view of the database. As we built on top of *TxCheck*, the instrumentation we use is very similar to the original one, with the addition of 3 new instrumentation types. The instrumentation types we use are:

- **Before Write Read (BWR)**: Before a write operation, a *SELECT* statement is added to read the version of overwritten objects.
- **After Write Read (AWR)**: After a write operation, a *SELECT* statement is added to read the new version of the overwritten objects.
- **Version Set Read (VSR)**: Before reads and predicated statements, a *SELECT* statement is added to read the version set of all objects in the used tables.

- **Before Predicate Match (BPM)**: Before a write operation, a *SELECT* statement is added for each predicate that appears in the test case, to read the objects and versions that match the predicate.
- **After Predicate Match (APM)**: Similar to the *Before Predicate Match*, but is inserted after the write operation.
- **Predicate Match (PM)**: Before an operation that contains a predicate, a *SELECT* statement reading the objects and versions that match the predicate is added.

This instrumentation has the potential of introducing unwanted locking behavior (i.e. the *SELECT* statements cause table locks), which is why we need to ensure that the instrumentation code is not interrupted by locking or other operations. In Figure 5.1, we show the instrumentation of a trivial test case.

5.6 Extracting the DSG with SQL Instrumentation

5.6.1 Methodology

Our tool is based on *TxCheck* [5], which we modify to suit our needs. Our dependency extraction process is thus similar to the one used in *TxCheck*, with slight modification, which are centered around anti-dependencies. As such, we will not go into detail about the direct read dependencies, direct item anti-dependencies, and direct item write dependencies, as they are similar to the ones used by Jiang, Z. [5], where a complete and sound technique for extracting them is described, and will focus on the direct predicate anti dependencies and predicate write dependencies.

Given a test case consisting of intertwined transactions, we do the following:

- Add instrumentation statements to the SQL queries.
- Ensure that the instrumentation code is not interrupted by locking or other operations.
- Run the test case, saving the results of the *SELECT* statements.
- Extract the DSG edges with the help of the instrumentation results.

The *Predicate Match*, *Before Predicate Match* and *After Predicate Match* statements are used to extract the *Overwrites*, which happen when a transaction overwrites the predicate of another transaction (see definition 5.1).

5.6.2 Extracting Overwrites

For any two statements A and B, where A is a write operation and B contains a predicate, we can check if A *overwrites* B if there exist a primary pk such

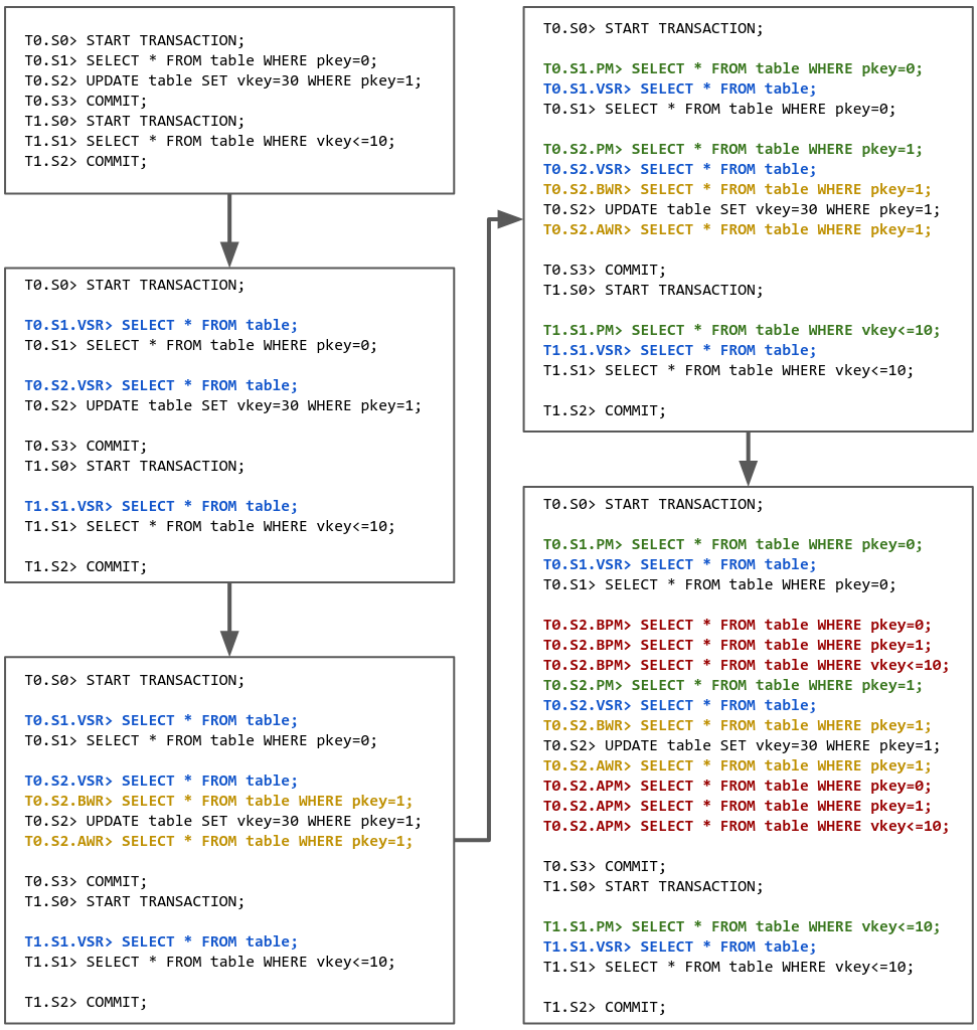

Figure 5.1: Instrumentation of a (simplified) test case.

that:

- The *After Write Read* instrumentation of A contains a record with the primary key pk and a version key v.
- The *Version Set Read* instrumentation of B contains a record with the primary key pk and a version v' such that v' happens before v in the list of versions for the primary key pk.
- The *Before Predicate Match* instrumentation of A for the predicate of B and the *Predicate Match* instrumentation of B contain pk, but the *After Predicate Match* instrumentation of A for the predicate of B does not, or the other way around.

Note that if B contains multiple predicates, we need to check for each

predicate if A overwrites B.

Proof We prove that our method for extracting overwrites is both sound and complete.

Soundness: If we flag A as *overwriting* B, then there exists a primary pk and two versions v and v' such that v is the version of pk installed by A, v' is the version of pk present in the version set of B, and v happens after v' in the list of versions for pk. Furthermore, the *Before Predicate Match* instrumentation of A for the predicate of B contains pk, but the *After Predicate Match* instrumentation of A for the predicate of B does not, or the other way around, meaning that A makes pk no longer match the predicate of B or the other way around. This is the definition of an *overwrite*.

Completeness: If A overwrites B, then there exists a record x for which A installs a version later than the one seen by B, and the two versions do not match the predicate of B. Let us denote the primary key of x as pk, the version installed by A as v, and the version seen by B as v'. Then, v happens after v' in the list of versions for pk, and the *Before Predicate Match* instrumentation of A for the predicate of B contains pk. However, the *After Predicate Match* instrumentation of A for the predicate of B does not, or the other way around. This means that our technique correctly identifies A as overwriting B. □

5.6.3 Extracting Predicate Write Dependencies

We proved that we are able to check if one statement overwrites another in a sound and complete manner. This allows us to trivially extract *direct predicate anti-dependencies* by checking if a write statement overwrites the predicate of a read statement.

Proof Let A be a write statement and B be a read statement. If A overwrites B, then the transaction containing A by definition *directly predicate anti-depends* on the transaction containing B. □

5.6.4 Extracting Direct Predicate Write Dependencies

In a similar way, we can extract the *direct predicate write dependencies* by checking that:

- The *After Write Read* of a transaction intersects with the *Version Set Read* of a predicated write statement of another transaction, or
- One transaction overwrites the predicate of another write transaction.

Proof We prove that our method for extracting direct predicate write dependencies is both sound and complete.

Soundness: If we flag A as *directly predicate write-depending* on B, then:

- The *After Write Read* of A intersects with the *Version Set Read* of B, and B is a predicated write. This means that A installs a version that is read by B, so A *directly predicate write-depends* on B, or
- A overwrites the predicate of B, so A *directly predicate write-depends* on B.

Completeness: If A *directly predicate write-depends* on B, then:

- A installs a version that is read by B, which means that the *After Write Read* of A intersects with the *Version Set Read* of B, so we flag A as *directly predicate write-depending* on B, or
- A overwrites the predicate of B, so we flag A as *directly predicate write-depending* on B. □

5.7 Implementation and Results

We implement our technique on top of *TxCheck*, which we modify to suit our needs. As we are only looking for isolation bugs, we do not need to run the single-transaction *TxCheck* uses as an oracle, so we bypass that part of the code.

The main changes we make to *TxCheck* are:

- Update the project to bring it to a working state: *TxCheck* relies on *Docker* images of *Ubuntu 20.04*, on which some dependencies of the project are not available. We fix the broken dependencies, and make the project runnable.
- Add the new *Before Predicate Match*, *After Predicate Match* and *Predicate Match* instrumentation types.
- Modify the dependency extraction code to create a complete and sound extraction of the DSG dependencies.
- Modify the bug minimization code to include support for the new instrumentation types.
- Add additional tests based on Adya's theory [9] for detecting isolation bugs, which were not previously possible due to the sporadic dependencies created by the original technique.

The fuzzing approach of our technique is similar to the one used by *TxCheck*, with the main difference being the ability to extract the DSG dependencies accurately, thus making the *graph de-cycling* phase of *TxCheck*, used to remove circular dependencies assumed to be false positives, unnecessary. This means that, while our approach and *TxCheck* share most of the implementation and work in a similar way, the two are quite different in the way they find bugs:

- Our approach relies on properties of the DSG to find isolation bugs.
- *TxCheck* checks a few properties of the DSG, but mainly relies on the single-transaction oracle to find bugs, as the DSG extraction used is not sound.

5.8 Evaluation

???

5.9 Limitations and Future Work

Our technique is a complete and sound method of extracting of the DSG dependencies, and a step forward in the direction of behavior-based DBMS testing. However, it suffers from similar limitations as the original technique used by *TxCheck*:

- The fuzzing is quite slow, with the test case generation and finding a non-blocking schedule being the bottleneck.
- While our technique is able to run under any of the isolation levels supported by the DBMS, the lower the isolation level the less checks we can perform, as the DSG has fewer restrictions. In particular, the *Read Uncommitted* isolation level imposes almost no restrictions on the DSG, which makes it hard to find bugs.
- Similarly to the original *TxCheck* technique, scheduling for some valid executions might fail due to the additional locking created by the instrumentation.

We only implemented support *SELECT*, *UPDATE* and *INSERT* statements. However, support for the *DELETE* statements is also possible. The main caveat is that *DELETE* removes entries from tables, while the Adya model assumes that deleted rows enter an *erased* state, in which they still exist belong in *Version Sets*. This can be solved by:

- Having a *deleted* copy of each table, initially empty.
- Before a *DELETE* statement, inserting the deleted rows in the *deleted* table.
- When performing the *Version Set Read* instrumentation, also reading the entries from the *deleted* table.

5.10 Conclusion

In this project, we improve the DSG extraction technique used by *TxCheck*, by adapting the SQL-level instrumentation technique introduced by Jiang, Z. [5]

to obtain a complete and sound list dependency. We implement the changes in the *TxCheck* project, giving *TxCheck* the ability to extract the DSG edges accurately.

While we did not have the time to run a full evaluation of our technique and to let the fuzzing run for a long time, we believe that our technique is a step forward in the direction of behavior-based DBMS testing, and that it has the potential to find more isolation bugs than the original technique used by *TxCheck*.

Chapter 6

Conclusion

During this project, we focused on the life-cycle of DBMS bug finding. We worked on multiple self-contained yet naturally related parts, namely:

- We developed a tool facilitating the replication of bugs in DBMSs, which can be used by bug reporters, developers or researchers for easily running test cases on multiple database systems and versions. The tool is available at https://github.com/theodormoroianu/MasterThesis, and released under the MIT license.
- We replicated 115 bugs, showcasing the usability of our tool, selected the bugs we considered to be interesting, and analyzed them.
- We built on top of the Adya dependency graph theory and its extraction using *SQL instrumentation*, and we proposed a complete and sound methodology for extracting the Adya dependency graph in a black-box manner, which is to the best of our knowledge the first of its kind. We also implemented this methodology on top of a fork of the *TxCheck* fuzzer, which is available at https://github.com/theodormoroianu/TxCheck, and released under the same license as the original project (GPL v3).

Bibliography

[1] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[2] L. A. Barroso and J. Clidaras, *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Springer Nature, 2022.

[3] Testim, *Unit test vs. integration test: What's the difference?* https://www.testim.io/blog/unit-test-vs-integration-test/, Accessed: 2024-10-07, 2022.

[4] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

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

[6] Z. Cui *et al.*, "Differentially testing database transactions for fun and profit," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[7] W. Dou *et al.*, "Detecting isolation bugs via transaction oracle construction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 1123–1135.

[8] Z. Cui *et al.*, "Understanding transaction bugs in database systems," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[9] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," 1999.

[10] *Mysql*, https://www.mysql.com/, Accessed: 2024-09-14.

[11] A. Akhtar, "Popularity ranking of database management systems," *arXiv preprint arXiv:2301.00847*, 2023.

[12] J. Gray *et al.*, "The transaction concept: Virtues and limitations," in *VLDB*, vol. 81, 1981, pp. 144–154.

[13] J. Melton, "Iso/ansi working draft: Database language sql (sql3)," *ISO/IEC SQL Revision. New York: American National Standards Institute*, 1992.

[14] J. Clark, A. F. Donaldson, J. Wickerson, and M. Rigger, "Validating database system isolation level implementations with version certificate recovery," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 754–768.

[15] *Podman*, https://podman.io/, Accessed: 2024-10-08.

[16] *Github copilot*, https://github.com/features/copilot.

[17] *Mysql test run*, https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN_PL.html, Accessed: 2024-10-08.

[18] *Innodb consistent read*, https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html, Accessed: 2024-11-06.

[19] *Tidb snapshot*, https://docs.pingcap.com/tidb/stable/system-variables, Accessed: 2024-11-14.

[20] *Innodb transaction isolation levels*, https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html, Accessed: 2024-11-15.

[21] *Mariadb transactions and isolation levels for sql server users*, https://mariadb.com/kb/en/mariadb-transactions-and-isolation-levels-for-sql-server-users/, Accessed: 2024-11-15.

[22] *Tidb transaction isolation levels*, https://docs.pingcap.com/tidb/stable/transaction-isolation-levels, Accessed: 2024-11-15.