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 analysing Database Management System (DBMS) transactional or isolation bugs, use this testbed to replicate and analyse a set of known bugs, and use the gained insights for developing a novel bug-finding tool.

First, we create a testbed leveraging containerisation technology to easily spin-up and run a variety of custom versions of DBMSs. The testbed provides an easy way of running concurent transaction workloads, and generates logs of the transactions 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 analyse a set of known bugs in the MySQL, MariaDB and TiDB DBMSs. We find that in a majority of the replicated bugs the isolation level does not have any impact, as bugs manifest under all isolation levels supported by the DBMS. We then analyse in depth the bugs manifesting under only a subset of the isolation levels.

Finally, we develop a novel bug-finding technique, which leverages dependencies graphs between transactions, computed using SQL-level instrumentation, for finding isolation bugs in DBMSs. We also implement this technique on top of 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.

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

Contents **iii**

1 Introduction **1**
1.1 Problems and Motivations 1
1.2 Contributions 2

2 Background **3**
2.1 Database Management Systems 3
2.2 Transactions 3
2.3 Isolation Levels 4
2.4 Transaction and Isolation Bugs 5
2.5 Previous Work 6

3 Developping a DBMS Transactional Testing Framework **7**
3.1 Overview 7
3.2 Design 7
3.3 Custom DBMS Version 8
3.4 Testing Meta-Language 9
3.5 Usage 11

4 Replicating Transactional Bugs in MySQL, MariaDB and TiDB **15**
4.1 Overview 15
4.2 Replicated Bugs 15
4.3 Analysis 16
4.4 Interesting Bugs 17
4.5 Conclusion 26

5 Detecting Isolation Bugs in DBMSs via Dependency Graphs Construction **27**
5.1 Introduction and Motivation 27
5.2 Data Model Used 28

5.3 DSG Dependencies 28
5.4 SQL-level Instrumentation 29
5.4.1 Intuition Behind SQL-level Instrumentation 29
5.4.2 Intrumentation Statements 30
5.4.3 Extracting Dependencies from Instrumented Queries . 30

Bibliography **33**

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 usecases 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 concurency 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 reseach.

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 existant DBMS transactional bugs reported to their respective issue trackers, and reported by or analysed in other works [5, 6, 7, 8], in order to undestand how they corelate 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 concurent 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 testing framework, levraging containerisation techniques for starting specific versions of DBMS servers, and automatically replicating DBMS bugs.
2. Using the testing framework, we replicate TODO bugs in the *MySQL*, *MariaDB* and *TiDB* DBMSs.
3. We analyse the reports of the replicated bugs, and we explore the corelation between isolation levels and the reported bugs.
4. We develop a novel black-box fuzzing technique, based on SQL instrumentation and Adya dependency graphs.
5. TODO: We implement the fuzzing technique, by modifying an existing fuzzing 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 developpers 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 consitency guarantees, at the cost of concurrency and performance. While many isolations 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.

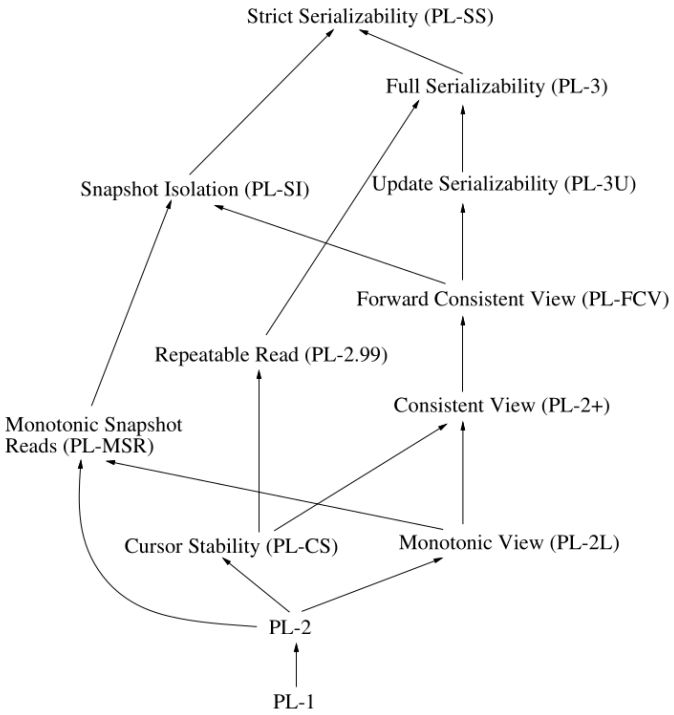

Figure 2.1: The Adya isolation levels [9].

In modern DBMSs, the isolation levels are implemented by levraging concurrency control techniques such as locking and multi-version concurrency control (MVCC). The choice of isolation level is a trade-off between consistency gurantees and concurency 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 dimishing 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 concurent transactions, might

occur sporadically due to the nondeterministic nature of the concurency, and might not be easily reproducable.

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 Previous Work

The topic of database testing is not new, with multiple techniques and tools being developed over the years. 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.

Our work is inspired by the surveying work of Cui, Z. et al. [8], and aims to replicate and analyze bugs, by providing an easy way to test them. In the best of our knowledge, the authors of the survey did not actually replicate the collected bugs, due to constrainst on the DBMS versions (often a Git commit), and time consumption, and relied on the original bug reports.

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, levraging the SQL instrumentation technique.

Chapter 3

Developping 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 analyse 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 developped 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 testcases, using a specific format, and is responsible for creating the internal representation of the testcases.
- 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.

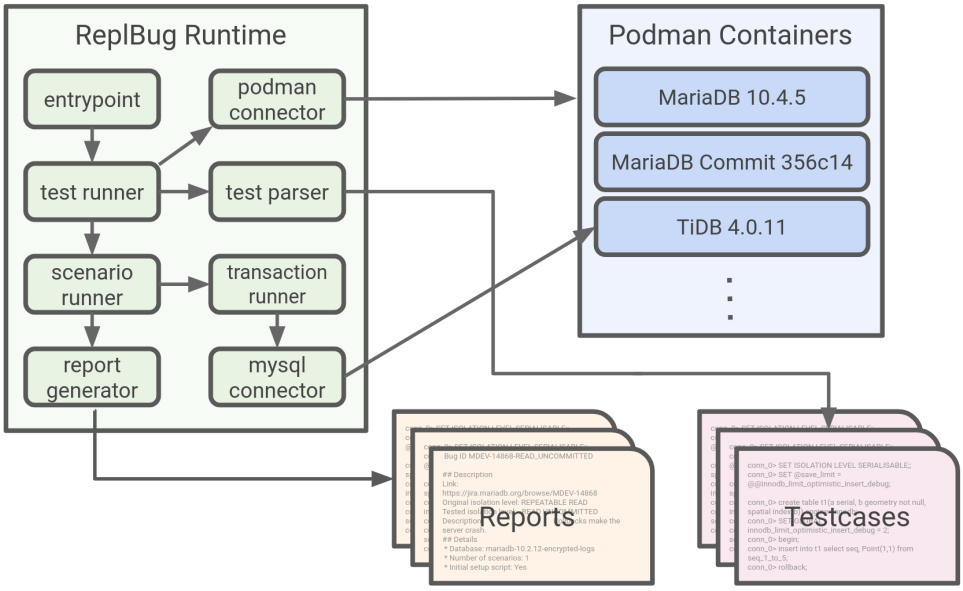

Figure 3.1: Design of the *ReplBug* testing framework

- 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 testcases under a specific configuration.
- The *test runner*: This component orchestrates the execution of all required testcases 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 specifig *Git* commit are not released as binaries.

For the mentioned reasons, we consider the ability to built DBMSs from source essential. To simplify the process, we provide sample *Dockerfile* templates, which can be used to test specific DMBS versions. A sample *Dockerfile* for *TiKV* 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 testcase, specifying the statements and their execution order on the DBMS is hard, due to multiple reasons:

- Transactional and isolation bug PoCs usually need multiple concurent 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 testcases 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 cubersome, 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 testcase for the bug *MDEV-26642* in *MariaDB 10.6.17*.

Each testcase provides the following information:

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

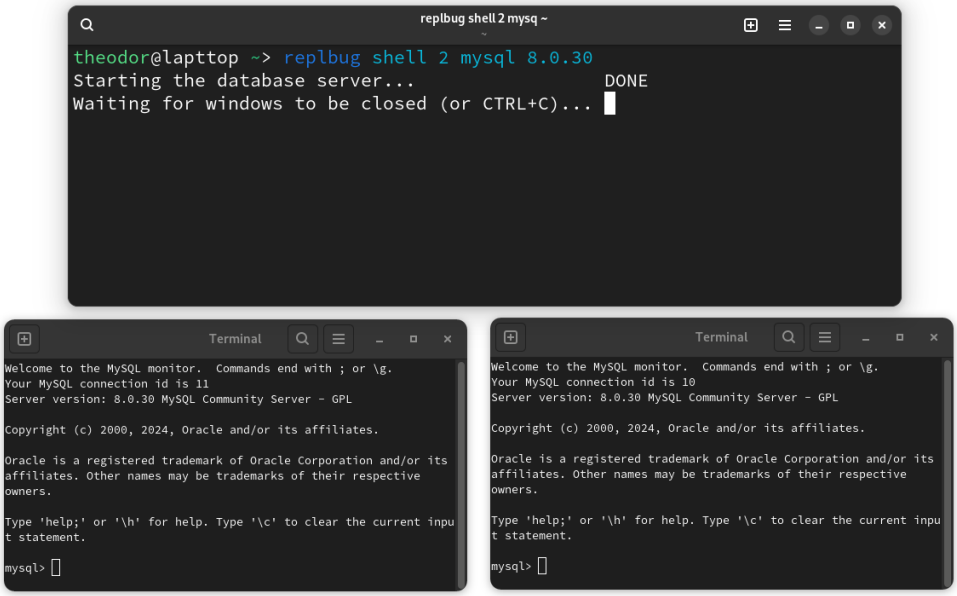

Figure 3.4: Using *ReplBug* to start 2 *MySQL v8.0.30* shells

- The bug ID and a link to the bug report.
- The setup script, which is executed before the testcases. 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 testcases that will be executed (one for each isolation level). Each scenario is a sequence of statements, executed in parallel by different connections.

For running a testcase, 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 analysed.

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.

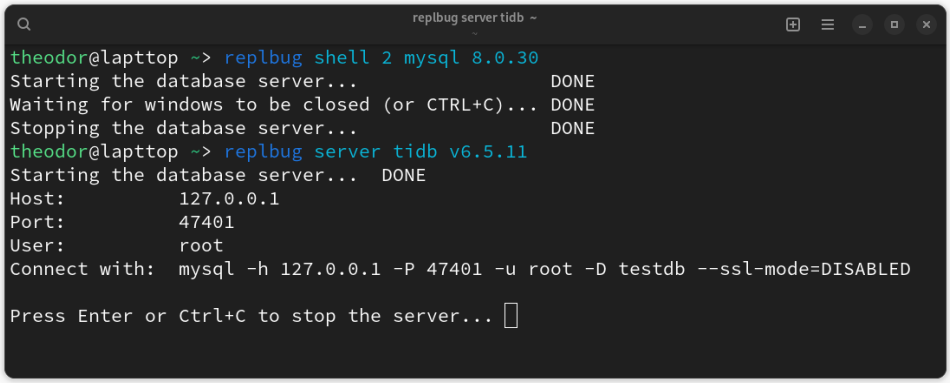

Figure 3.5: Using *ReplBug* to start a *TiDB v6.5.11* server

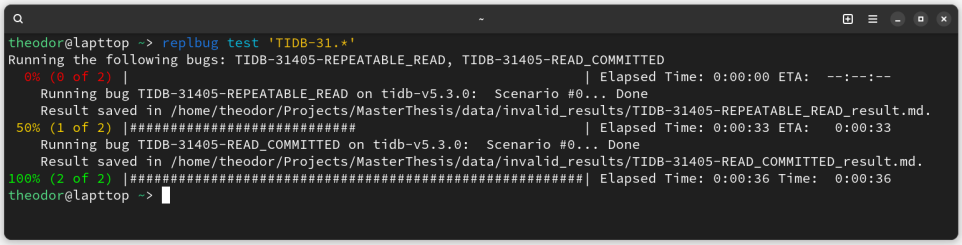

Figure 3.6: Using *ReplBug* to generate reports of some known bugs

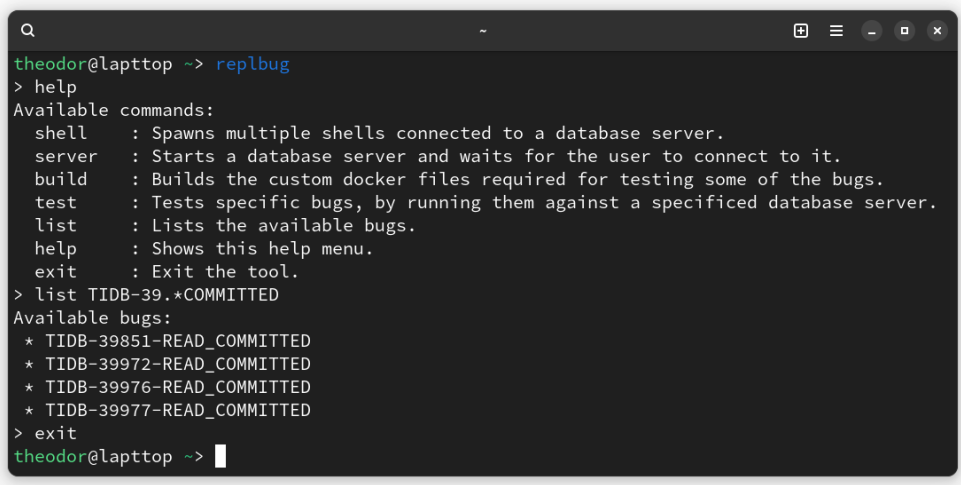

Figure 3.7: Using *ReplBug* in interactive mode

- `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 testcases 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.

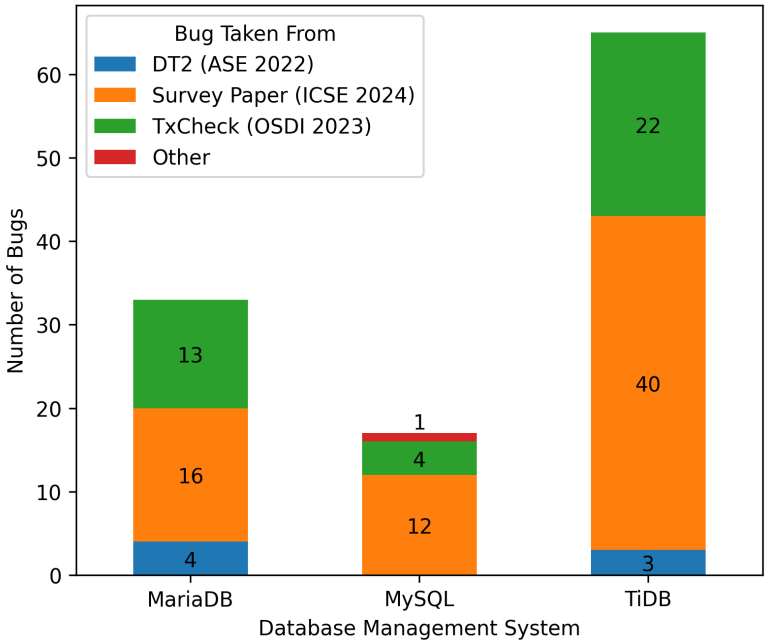

Figure 4.1: Distribution of bugs by DBMS and reporting paper [8, 7, 6].

- We write a testcase in the *ReplBug* testing framework, using the meta-language described in the previous chapter.
- We run the testcase on the DBMS, under all supported isolation levels.
- We sometimes have to adjust the testcase, 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 analyse the output of the testcases. 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

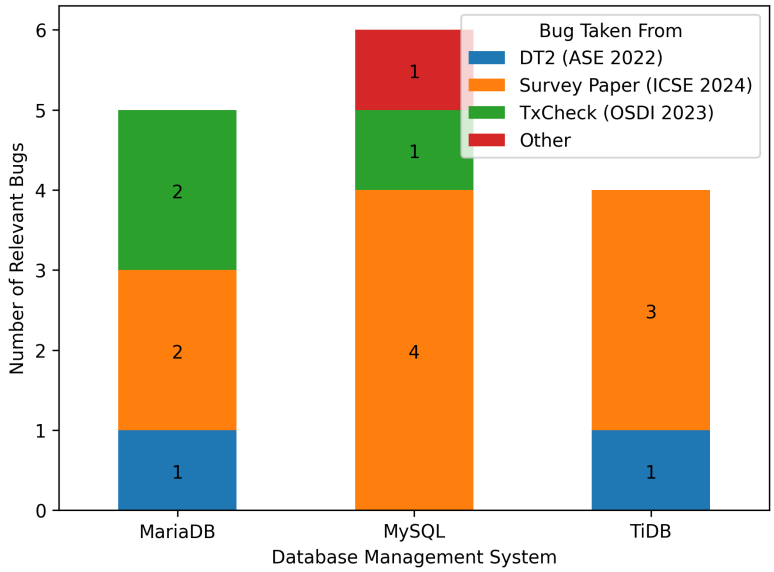

Figure 4.2: Relevant bugs by DBMS and reporting paper [8, 7, 6].

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 *intersting 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 behaviour, where possible. The bugs are presented in

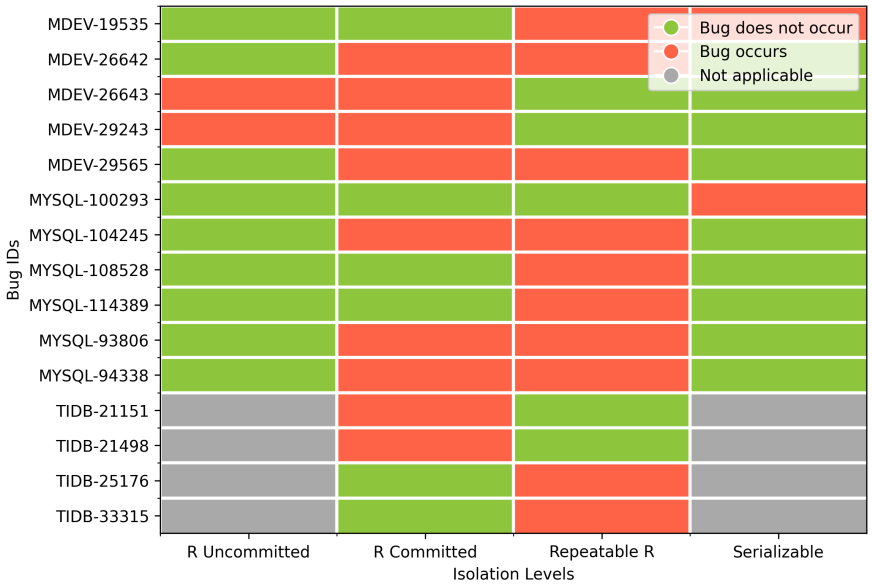

Figure 4.3: Relevant bugs by isolation levels.

the 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 behaviour of other DBMSs.

When the `sql_mode` is set to `ORACLE`, *MariaDB* ommits to add exclusive locks when running an *SELECT FOR UPDATE* statement. This leads to incorrect behaviour 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 concurent 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 retrival 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 intented behaviour, 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), *Serialisable* 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*.

Wen 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 intented. The PoC is simplified in Figure 4.7.

In the PoC, one transaction updates a table and commits. Another concurent 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 behaviour is similar to the behaviour of *MDEV-29565*, and we consider it is due to the same poorly-documented feature [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*). It was closed as *duplicate*, but the original bug is still open.

As the bug is still open, we do not know the root cause of the bug, whose PoC 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 misshandling 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 diry-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` function when required.

```
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 concurent 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 *Repetable 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 concurent 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 behaviour.

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 behaviour 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 tradeoff between performance (under the form of concurrency) and isolation.

Modern database systems usually express isolation guarantees under a *locking behaviour*, 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 behaviour 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, requiering 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 neccessarily *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 inherintly 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].

In this chapter, we introduce a novel bug-finding technique, which leverages Direct Serialization Graphs (DSG) introduced by Adya, A. [9] and SQL-level instrumentation introduced by Jiang, Z. [5]. This technique is a modification of the one used by Jiang, Z.. We also implement a tool based on our technique on top of *TxCheck*, which we modify to suit our needs.

5.2 Data Model Used

We consider Adya's data model, which is slightly different from the standard data model used in the SQL databases. In Adya's model, a transaction T_i is a sequence of operations read and write operations, with or without a predicate. Each transaction installs new versions of objects, and reads or writes versions of objects installed by other transactions. We also consider the history model introduced by Adya, A. [9].

In his work, Adya, A. [9] introduces the concept of Direct Serializability Graphs (DSG), in which nodes are transactions and edges are directed dependencies or anti-dependencies between transactions.

In this project, we build the DSGs using SQL-level instrumentation, and we use them to detect isolation bugs in DBMSs by searching for cycles, which Adya proved to be impossible in a corect DBMS [9].

5.3 DSG Dependencies

We take the following definitions and notations from Adya, A. [9]:

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 instaleld 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-anty-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.*

Our end goal is to find all *Direct Read Dependencies*, *Direct Write Dependencies* and *Direct Anti-Dependencies* between transactions, to build the DSGs using these dependencies and check for cycles, which can only occur due to an isolation bug in the DBMS.

5.4 SQL-level Instrumentation

5.4.1 Intuition Behind SQL-level Instrumentation

For finding the DSG edges (dependencies between transactions), we use SQL-level instrumentation, introduced by Jiang, Z. [5]. We instrument the SQL queries by adding SQL instrumentation code right before and right after

each SQL query, and ensuring the instrumentation code is not interupted by locking or other operations, and ensure that every table of the database stores a *version* column, which stores the version of the object.

The informal intuition behind SQL-level instrumentation is that we wish to query the current state of the database, but lack the white-box access to the internal state of the DBMS (which is how Clark, J. et al. [14] extract the DSGs). As we can only query the database, we infer the dependencies by injecting *SELECT* statements in testcases, in order to be able to construct a sound and complete DSG.

5.4.2 Intrumentation Statements

We intrument the SQL queries by adding the following statements, with slight variations from Jiang, Z. [5]:

- **Before Write Read**: Before a write operation, we add a *SELECT* statement to read the current version of overwritten objects.
- **After Write Read**: After a write operation, we add a *SELECT* statement to read the new version of the overwritten objects.
- **Version Set Read**: Before reads and predicates, we add a *SELECT* statement to read the version set of all objects in the used tables.
- **Before Predicate Match**: Before a write operation, we add a *SELECT* statement for each predicate that appears in the generated statement, to read the objects and their versions that match the predicate.
- **After Predicate Match**: Similar to *Before Predicate Match*, but after the write operation.
- **Predicate Match**: Before an operation that contains a predicate, we add a *SELECT* statement to read the objects and their versions that match the predicate.

5.4.3 Extracting Dependencies from Instrumented Queries

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. Given a testcase 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 testcase, saving the results of the *SELECT* statements.

- Extract the DSG edges with the help of the instrumentation results.

To extract the DSG edges from the instrumented queries, we use rules similar to the ones used by Jiang, Z. [5], with slight modifications. We first extract dependencies on a statement level, and then combine them to extract dependencies on a transaction level (the DSG).

Direct Read Dependencies

We extract the *Direct Item-Read* dependencies by checking if the *After Write Read* of the write statement and the output of the read statement intersect.

Proof Let T_i be a transaction that writes an object x_i and T_j be a transaction that reads x_i. The *After Write Read* of T_i contains x_i, and the output of the read statement of T_j contains x_i. Thus, the two intersect. Similarely, if T_i writes does not write a version of x that T_j reads, the two do not intersect.□

We extract the *Direct Predicate-Read* dependencies by checking if the *After Predicate Match* of the write statement and *Version Set Read* of the read statement intersect.

Proof Let T_i be a transaction that writes an object x_i and T_j be a transaction containing a predicate including x_i in its version set.The *After Predicate Match* of T_i contains x_i, and the *Version Set Read* of T_j contains x_i. Thus, the two intersect. Similarely, if T_i writes does not write a version of x included in T_j's version set, the two do not intersect. □

Direct Anti-Dependencies

We extract the *Direct Item-Anti-Depends* by:

- Extracting a list of the versions of each object, ordered by the installation order.
- Checking if the version of the object read by the read statement is earlier than the version of the object read by the *After Write Read* of a write statement.

We extract the *Direct Predicate-Anti-Depends* by:

- Extracting a list of the versions of each object, ordered by the time of installation.
- Checking if the version of the object set by the write statement is later than the version of the object in the version set of a predicated read statement.

- Checking if the object's match status changes between the two versions, by using the *Before Predicate Match* and *After Predicate Match* statements.

Direct Write Dependencies

We extract the *Direct Item-Write* dependencies by checking if the *After Write Read* of the write statement and the *Before Write Read* of the next write statement intersect.

We extract the *Direct Predicate-Write* dependencies by checking if:

- The *After Write Read* of a transaction intersects with the *Version Set Read* of a predicated write statement of another transaction, and
- Checking the same condition as for the *Direct Predicate-Anti-Depends*, but with a write statement instead of a read.

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.