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

5 Detecting Isolation Bugs in DBMSs via Dependency Graphs Construction **17**
5.1 Introduction 17
5.2 Data Model Used 17
5.3 DSG Dependencies 17
5.4 SQL-level Instrumentation 18
5.4.1 Intuition Behind SQL-level Instrumentation 18
5.4.2 Intrumentation Statements 19

5.4.3 Extracting Dependencies from Instrumented Queries . 19

Bibliography **23**

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.

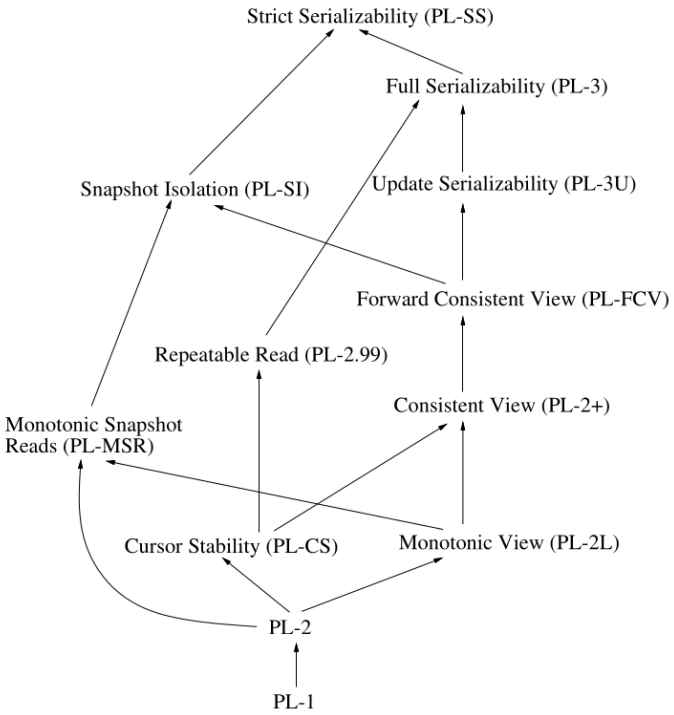

Figure 2.1: The Adya isolation levels [9].

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.

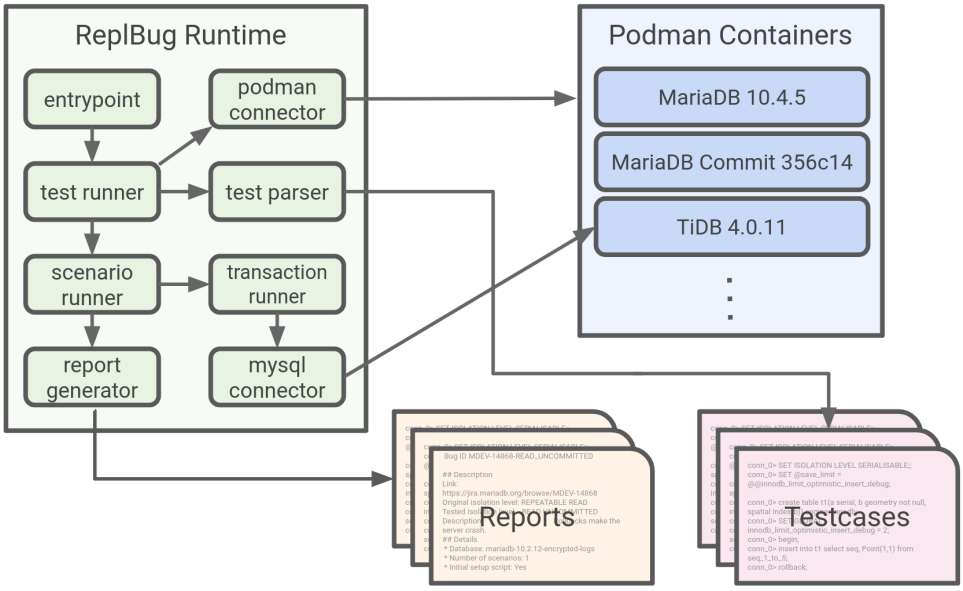

Figure 3.1: Design of the *ReplBug* testing framework

- 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.

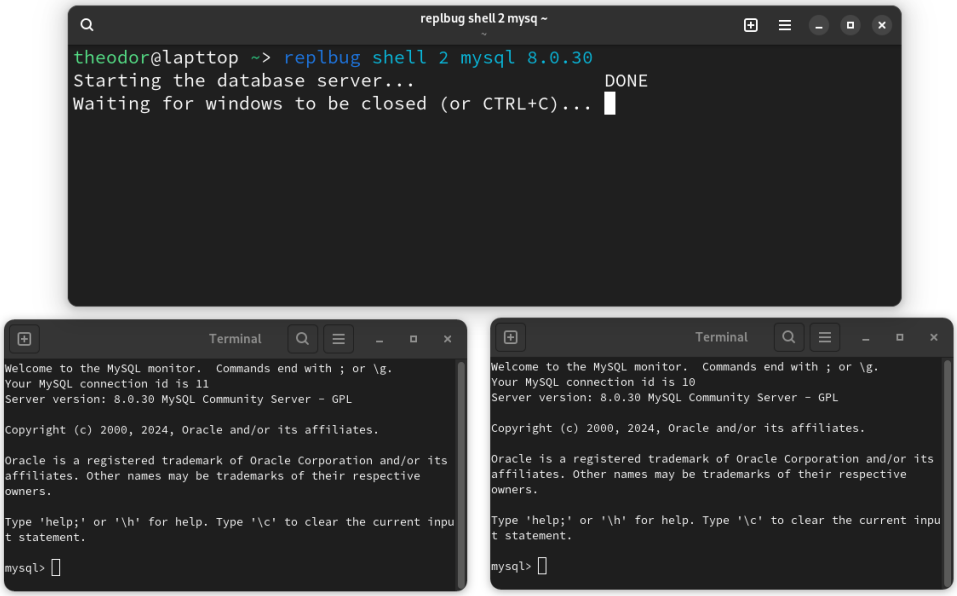

Figure 3.4: Using *ReplBug* to start 2 *MySQL v8.0.30* shells

- 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.

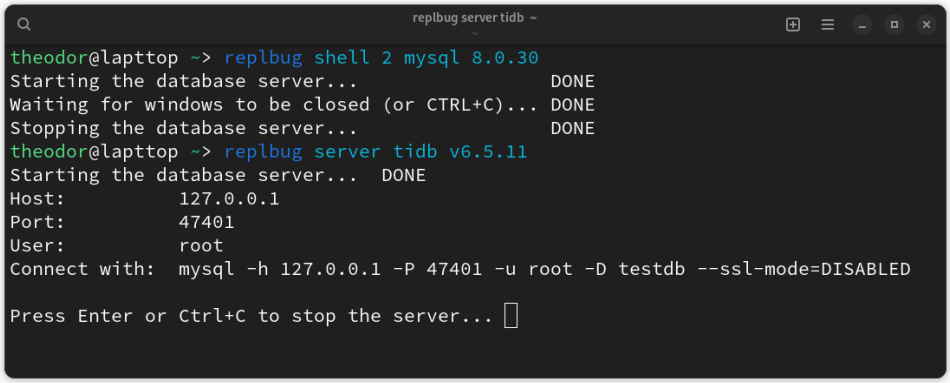

Figure 3.5: Using *ReplBug* to start a *TiDB v6.5.11* server

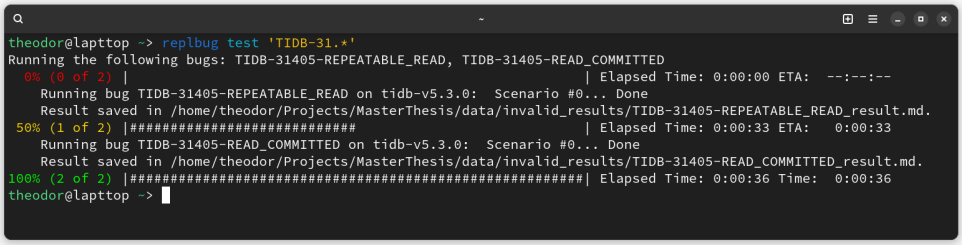

Figure 3.6: Using *ReplBug* to generate reports of some known bugs

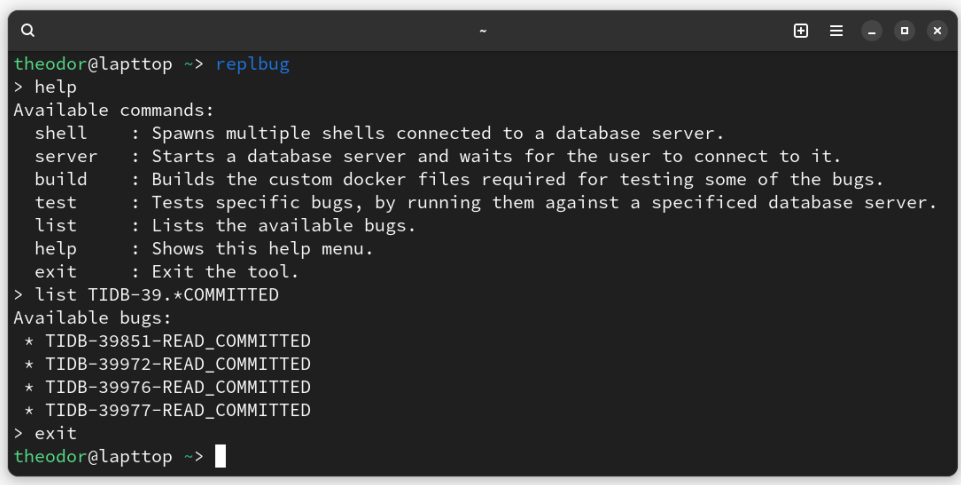

Figure 3.7: Using *ReplBug* in interactive mode

- `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.

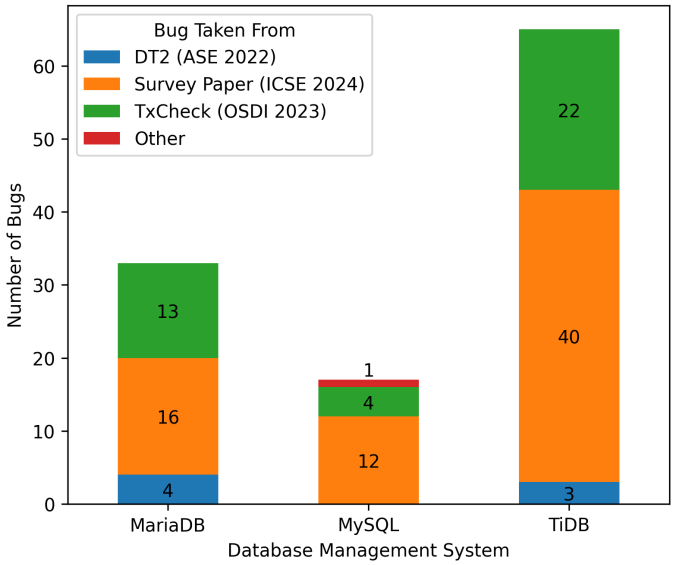

Figure 4.1: Distribution of bugs by DBMS and reporting paper [8, 7, 6].

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*).

Chapter 5

Detecting Isolation Bugs in DBMSs via Dependency Graphs Construction

5.1 Introduction

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].

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.