ABSTRACT

SHAO, SHUDI. Characterizing Performance Bugs, Deadlocks, and Exception Handling Bugs in Database-Backed Web Applications. (Under the direction of Guoliang Jin and Ruozhou Yu.)

Nowadays, database-backed web applications, such as those for online shopping and social networking, are widely used in people's daily lives. While there are many factors contributing to the success of a web application, performance and reliability are among the most crucial. However, software bugs, such as performance bugs, deadlocks, and exception handling bugs, are huge obstacles to the assurance of performance and reliability. These bugs can do a lot of harm to user experience and even lead to huge economic losses. With the wide availability of hand-held devices, much more people can easily access web applications, thus these bugs become more critical as they can occur more often with the increasing user base. In this dissertation, we conduct comprehensive characteristic studies on real-world performance bugs, deadlocks, and exception handling bugs to help developers better understand and combat these issues when developing database-backed web applications.

Our dissertation starts with characterizing performance bugs. While much research work has explored database-access antipatterns in database-backed web applications, these studies either did not focus on the performance aspect or only studied real-world performance bugs from ORM-based web applications, ignoring another major type of web applications, i.e., raw-SQL web applications. To address these limitations, we first summarize and report 24 known database-access performance antipatterns through a literature survey. We then collect performance bugs from raw-SQL web applications to check how extensively the known antipatterns can cover these bugs. To this end, we extract 10 new database-access performance antipatterns based on bugs that are not covered by the known performance antipatterns. Our results help to improve the coverage of database-access performance antipatterns.

In addition to performance bugs, we also conduct a study on deadlocks in database-backed web applications. Most existing work on deadlocks only focused on multi-threaded programs and did not cover database-lock deadlocks in web applications. For those few detection techniques for web-application deadlocks proposed by the software engineering community modeled the database locking behavior conservatively, potentially missing many real-world deadlocks. While database engines provide error logs for database-lock deadlocks, it is difficult for developers to understand them as database locks are usually

implicitly acquired and require knowledge of database internals. Given these limitations, we characterize deadlocks from real-world web applications. Specific to database-lock deadlocks, we categorize them into four hold-and-wait cycle patterns based on different database lock types involved in each deadlock. We believe these patterns will be useful for developers to understand, diagnose, and fix database-lock deadlocks.

Finally, we present a characteristic study on exception handling bugs. Modern database-backed web applications are usually built on top of programming languages that normally provide a built-in mechanism for exception handling. The exception handling mechanism allows web applications to continue running when an error occurs during runtime. However, failing to catch or handle exceptions properly leads to exception handling (EH) bugs. To date, no existing study has studied EH bugs in database-backed web applications. Therefore, we conduct a comprehensive characteristic study on 214 real-world EH bugs, including 199 no exception handling bugs and 15 improper exception handling bugs. We derive 6 main exception types from these EH bugs. We also characterize root causes and fixing strategies for no exception handling bugs and improper exception handling bugs, respectively.

In summary, this dissertation characterizes real-world performance bugs, deadlocks, and exception handling bugs that are collected from popular open-source database-backed web applications. We believe that our characteristic results can provide developers with useful guidance for testing, diagnosing, and fixing these bugs. We also expect that our characteristic results can guide researchers and tool vendors in designing and developing tool support to combat these bugs.

Characterizing Performance Bugs, Deadlocks, and Exception Handling Bugs in Database-Backed Web Applications

by
Shudi Shao

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2024

APPROVED BY:

Rada Chirkova	Alexandros Kapravelos
Xu Liu	Christopher Parnin
Guoliang Jin Co-chair of Advisory Committee	Ruozhou Yu Co-chair of Advisory Committee

BIOGRAPHY

Shudi Shao was born in Liuhe, a small town in Jiangsu Province, China, where he spent most of his childhood and teenage years. He earned both of his Bachelor's degree and Master's degree from East China University of Science and Technology in 2013 and 2016, respectively. After that, he decided to pursue his Ph.D. degree at North Carolina State University, under the guidance of Dr. Guoliang Jin and Dr. Ruozhou Yu. His research interest lies in exploring different issues in real-world database-backed web applications, such as performance bugs, concurrency bugs, and exception handling bugs.

ACKNOWLEDGEMENTS

It has been eight years since I began my Ph.D. studies in 2016. Although it has taken a considerable amount of time to complete my work, I have never regretted it and have enjoyed every moment of this journey. Reaching this point would not have been possible without the immense encouragement and support of many wonderful people.

First and foremost, I would like to thank my advisor, Prof. Dr. Guoliang Jin, for advising me during my whole Ph.D. study. Along the way to my final dissertation, I encountered numerous problems, difficulties, confusions. But no matter how hard it was, Dr. Jin always supported me with kindness and patience. I remember every moment when he stayed up late together with me to catch the deadlines and refine paper writings. I also remember every moment when he helped me with polishing slides and practicing presentations again and again with valuable comments and suggestions. Thanks to him, I gained the ability to build up the critical thinking way and to effectively communicate via oral presentations. Most importantly, he taught me to learn to accept failure and try again.

I would also like to thank Dr. Ruozhou Yu. I'm blessed to have had him as my co-advisor after Dr. Jin decided to pursue a career path in industry. Whenever I sent a message or an email to him to ask for help, he always responded in the first time. Not to mention, his suggestions on scientific research and paper writing are invaluable in completing this thesis.

I am grateful to my committee members, Dr. Alexandros Kapravelos, Dr. Xu Liu, Dr. Chris Parnin, and Dr. Rada Chirkova, for their insightful opinions and technical guidance on my study. Especially, I want to express my gratitude to Dr. Xu Liu for serving as my interim co-advisor during my oral preliminary exam.

I truly appreciate Dr. George Rouskas for his assistance and suggestions provided in dealing with all kinds of paperwork during my whole Ph.D. study.

I also want to thank my former labmates, Qi Zhao and Dr. Zhengyi Qiu. I have received limitless help and support from them. The time we spent together on having research discussions is precious.

As for my personal life, I first want to express special thanks to Weijia Li, who was the first friend I made at NCSU. Although he is no longer with us, his support, perseverance, and optimism will always inspire me. I also would like to thank my friends and former roommates, Dr. Fan Zhang, Dr. Tianpei Xia, Dr. Xi Yang, Dr. Zhengwang Wu, Dr. Jingzhu He, Dr. Peipei Wang, Dr. Rui Shu, Dr. Zifan Nan, Zhiren Lu, Jialin Cui, Hao Zhang, Lei Huang, and Xiaoyu Chen, for their care and encouragement. I also want to thank everyone who has helped me on my way to completing the Ph.D. program.

Finally, I would like to express my gratitude to my parents, Jianping Zhu and Daiying Shao, my aunt, Lianying Shao, my godmother, Ying Chen, and all other family members for their unconditional love and support!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Background and Motivation

Nowadays, modern web applications, like online shopping and social networking, are widely used in people's daily lives. These web applications usually have a three-tier architecture as shown in figure 1.1. The first tier is the frontend, which is a web interface on the browser side that renders context and images. It is usually developed by a markup languages like HTML. Then, in the middle, it is the application logic tier on the server side, which is usually developed via different kinds of object-oriented programming languages, such as Java, Ruby, PHP, and etc. In the last, the database is usually used as a backend to store and manage user-generated data. When a web application runs, and a user visits the website and clicks on a button or link on the webpage, the front-end will issue a request to an application server. After that, the middle tier, i.e., the application logic code will process the request by its corresponding request handler. During this process, the middle tier sends SQL queries to the database to retrieve or modify the stored data. With the results returned from the database, the application logic code will then construct a response to send those results back to the frontend. Finally, the frontend receives the response and renders the results to the user in the browser. This is the whole general flow of how a database-backed web application works.

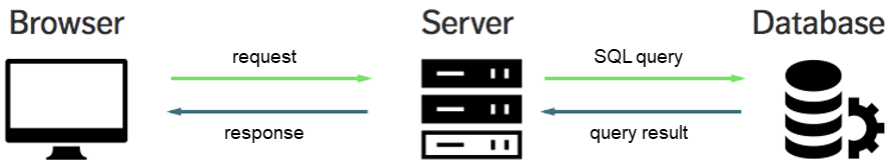

Figure 1.1: A high-level architecture of the database-backed web applications

Under such an architecture, especially with the application logic tier and the database backend tier, it is challenging for developers to avoid performance bugs, deadlock bugs, and exception handling bugs while implementing a database-backed web application:

- On performance, developers need to be familiar with queries and how a database engine executes queries so that they can write optimized queries. They also need to be cautious when designing and implementing application logic to avoid generating unnecessary round trips between the tiers of database-backed web applications. What is worse, performance issues are difficult to be exposed during testing, as they might only occur with large amounts of data that may only be generated after production deployment.
- On deadlocks, it is challenging for developers to understand database-lock deadlocks because database locks are usually implicitly acquired when executing queries. This requires developers to have knowledge of database internals. Besides, database-lock deadlocks are also difficult to be triggered during testing, as they might only occur in high-concurrency scenarios.
- On exception handling, modern web applications can have very complex business logic as they may contain many web pages and each page can involve various functionalities. This complexity increases the number of conditions where potential exceptions might arise. It is difficult for developers to take all of the exception conditions into consideration and handle them properly, thus leading to exception handling bugs.

Detecting and resolving performance bugs, deadlocks, and exception handling bugs is the key to the success of a web application. However, current studies fail to provide a comprehensive understanding of these aforementioned bugs. In this thesis, we will characterize real-world performance bugs, deadlocks, and exception handling bugs that are collected from popular open-source database-backed web applications to help developers

better understand, diagnose, and fix those bugs they may encounter. We also expect that our characteristics results can guide researchers and tool vendors to design and develop tool support to combat these bugs.

1.2 Summary of Contributions

In this dissertation, we make the following contributions:

- **Database-Access Performance Antipatterns in Database-Backed Web Applications.** Database-backed web applications are prone to performance bugs related to database access. While much work has been conducted on database-access antipatterns (commonly used solutions to recurring problems that generate negative consequences), and some recent work has focused on performance impact, there still lacks a comprehensive view of database-access performance antipatterns in database-backed web applications. To date, no existing work systematically has reported known antipatterns in the literature, and no existing work has studied database-access performance bugs in raw-SQL web applications, which embed SQL queries into the application code via language-provided SQL interfaces. To address this gap, we first summarize all known database-access performance antipatterns found through our literature survey in a total set of 47 papers and 1 book. We further collect 140 database-access performance bugs from popular real-world open source raw-SQL web applications, which have been largely ignored by recent work, to check how extensively the known antipatterns can cover these bugs. For bugs not covered by the known antipatterns, we extract new database-access performance antipatterns based on these real-world performance bugs from such web applications. Our study in total reports a total of 24 known and 10 new database-access performance antipatterns. This work was published in ICSME 2020 [150].

- **A Characteristic Study of Deadlocks in Database-Backed Web Applications.** By the nature of web applications, they are concurrent and thus subject to deadlocks. In web applications, the core business logic is a group of request handlers, which are responsible for handling incoming HTTP requests. Deadlocks in database-backed web applications could involve different numbers of HTTP requests. They also could be caused by locks explicitly requested in application code or implicitly requested by databases during query execution. To have a comprehensive understanding of real-world deadlocks in web applications, we conduct a characteristic study with

49 deadlocks collected from real-world web applications developed following different programming paradigms. We provide categorization results based on HTTP request numbers and resource types, with a special focus on categorizing deadlocks on database locks. We identify 36 deadlock bugs on database locks from real-world web applications. To further complement our understanding of deadlock patterns, we also search questions on StackOverflow for analysis and obtain a set of 27 questions. From these collected bugs and question posts, we summarize four patterns of deadlocks on database locks that differ on the types of resources involved in deadlock hold-and-wait cycles. Besides, we also categorize the fixing strategies used for these bugs. This work was published in ISSRE 2021 [135].

- **A Characteristic Study of exception handling bugs in Database-Backed Web Applications.** Modern database-backed web applications are usually built on top of object-oriented programming languages, which normally provide a built-in mechanism to catch and handle raised exceptions during runtime. The exception handling mechanism allows web applications to continue running when an exception occurs. Failing to catch or handle exceptions properly can lead to issues such as page crash and negative user experience. We refer to these bugs as exception handling (EH) bugs. To date, no existing study has studied exception handling bugs in database-backed web applications. To help developers better understand and combat exception handling bugs, we conduct the first comprehensive characteristic study on 214 real-world exception handling bugs across seven popular open-source web applications written in Ruby-on-Rails, which is a widely used Object-Relation-Mapping (ORM) framework. We find that most of the exception handling bugs (199 out of 214) are caused by failing to catch an certain exception while only 15 bugs are caused by improper exception handling. We derive six main exception types from the collected exception handling bugs. We also characterize root causes and fixing strategies for no exception handling bugs and improper exception handling bugs, respectively. We expect that our results can help developers avoid exception handling bugs when developing applications, and guide the design and development of tools for detecting and automatically fixing these bugs.

1.3 Structure of the Thesis

The remainder of the thesis is organized as follows:

- Chapter 2 describes our empirical study of database-access performance antipatterns

in database-backed web applications.

- Chapter 3 presents our understanding of deadlocks in database-backed web applications.
- Chapter 4 introduces our characterization results on real-world exception handling bugs in web applications.
- Chapter 5 explores opportunities to design tool support for performance bugs, deadlocks, and exception handling bugs based on our study results.
- Chapter 6 discusses the related work.
- Chapter 7 concludes the thesis and proposes potential directions for future research.

CHAPTER

2

DATABASE-ACCESS PERFORMANCE ANTIPATTERNS IN DATABASE-BACKED WEB APPLICATIONS

2.1 Introduction

Despite high criticality of performance bugs related to database accesses, it is challenging to develop database-backed web applications that interact with backend databases efficiently. As critical performance information and constraints of database accesses are hidden behind the database-access interface, developers do not necessarily have good knowledge on the performance implication of various database accesses and the interaction between database accesses and application code at development time. Therefore, database-access-related performance bugs in database-backed web applications are challenging to be avoided or identified.

To handle performance bugs especially those related to database accesses, researchers have conducted empirical studies to understand their common characteristics or abstract their belonging antipatterns. Some studies focus on performance bugs in general server and client applications [92, 128], Android applications [113], and client-side JavaScript

programs [148]. Since these performance bugs being studied are not specific to server-side database-backed web applications, the gained insights cannot be directly generalized to web-application performance bugs related to database accesses. Specific to database-backed web applications, while database-access antipatterns have been studied for a long period of time, it is only recently that previous work [66, 118] started to focus specifically on antipatterns with performance impact that can lead to performance bugs.

However, the previous work still fails to provide a comprehensive understanding of database-access performance antipatterns in web applications.

- To summarize known database-access performance antipatterns, researchers [66, 118] conducted literature surveys, but they did not comprehensively cover or report antipatterns that exist in the literature. Chen et al. [66] surveyed five papers [68, 70, 71, 163, 166] and reported 17 antipatterns that can be used for performance-aware refactoring. While the five surveyed papers all have a focus on performance issues related to database accesses in web applications, Chen et al. did not include many more related papers and books that exist in the literature. Lyu et al. [118] surveyed 56 research papers and 1 book with database-access antipatterns, not all of which are related to performance, and they reported 8 performance antipatterns and 3 security antipatterns. While Lyu et al. surveyed more papers and books, they unfortunately did not report all performance antipatterns from the surveyed literature. Their paper does not describe the reason why some antipatterns were excluded. One guess could be that their focus was on how mobile applications use local databases and they excluded performance antipatterns that were not found under this context.

- To extract new database-access performance antipatterns, researchers studied real-world performance bugs in bug-tracking systems [166] and/or performance issues found by static analysis or dynamic profiling tools [66, 163, 166], but they did not cover comprehensive types of database-backed web applications. Recent characteristic studies focused on performance bugs in web applications developed using different Object Relational Mapping (ORM) frameworks, including ActiveRecord for Ruby on Rails [163, 166] and Eloquent for PHP [66]. However, there are still a lot of web applications that access databases by directly constructing queries with language-provided SQL interfaces in application code but not through ORM frameworks, and we refer to these applications as *direct-accessing web applications*. Notably, among the top 15 most popular websites [37], MediaWiki and WordPress are open sourced, and neither of them uses an ORM framework. It is not clear whether known performance antipatterns in the literature can cover comprehensively performance bugs in direct-

accessing web applications.

To complement the current state of the art, in this chapter, we first conduct a literature review to summarize the current knowledge about database-access antipatterns that can lead to performance bugs, and our goal is to cover and report known antipatterns in the literature as comprehensively as possible. After gathering known antipatterns, we conduct a characteristic study of performance bugs in direct-accessing web applications to investigate whether there are new database-access performance antipatterns not covered by the existing research literature.

Specifically, we address the following three major research questions in this chapter:

- **RQ1:** What are the known database-access performance antipatterns in the research literature?
- **RQ2:** How extensively can known antipatterns cover the root causes of performance bugs in direct-accessing web applications?
- **RQ3:** Are there any unknown database-access performance antipatterns based on performance bugs in direct-accessing web applications?

To answer **RQ1**, we follow the literature-survey methodology laid out by Lyu et al. [118]. Since the list of publications studied by Lyu et al. is not available in their paper or on their GitHub site, where the benchmark suite and raw measurement data are available at the time of checking, we repeat the literature search process. In total, from our surveyed research publications, we identify 24 performance antipatterns, which are substantially more than those identified via the previous two literature surveys [66, 118]. By listing all the studied papers and reporting all the known antipatterns, future research does not have to repeat the same literature-survey process again and can reuse our results as the starting point.

For **RQ2** and **RQ3**, we study 140 database-access related performance bugs collected from the bug-tracking systems of seven direct-accessing web applications, i.e., *BugZilla*, *DNN*, *Joomla!*, *MediaWiki*, *WordPress*, *Moodle*, and *Odoo*. We make our subject collection based on popularity of applications, availability of bug-tracking systems, and sufficiency of information for bug understanding. Our selection covers four different popular web-application languages, i.e., PHP, C#, Python, and Perl.

To answer **RQ2**, we use the 24 known performance antipatterns identified while answering **RQ1** to match with the root causes of our collected performance bugs. Among the 140 collected performance bugs, 107 of them match with known performance antipatterns.

To answer **RQ3**, we further extract 10 new performance antipatterns from the 33 performance bugs identified while answering **RQ2**, which cannot be covered by known performance antipatterns. Among these 10 new antipatterns, only 1 antipattern is related to query construction and thus specific to direct-accessing web applications, and the other 9 are all applicable to database-backed web applications in general. Our results confirm the necessity of studying direct-accessing web applications to provide a more comprehensive coverage on real-world database-access performance antipatterns.

2.2 Methodology

In this section, we describe our methodology on (1) how we survey the existing literature for summarizing known database-access performance antipatterns to answer **RQ1** and (2) how we collect and analyze performance-bug reports from popular direct-accessing database-backed web applications to answer **RQ2** and **RQ3**.

2.2.1 Literature Search Protocol

We follow the literature search protocol (as described by Lyu et al. [118]) developed based on Kitchenham's systematic review guidelines [102]. Specifically, we start with 8 papers [53, 66, 68, 70, 71, 118, 163, 166] as our initial set and conduct forward/backward citation search with the initial set. During the process, we include papers that discover or define at least one antipattern related to database accesses and add them to our set. We stop when the forward/backward citation search results in only papers that are already included.

In total, we obtain 47 papers and 1 book, while Lyu et al. obtained 56 papers and 1 book [118]. As we include selected papers published in only recent high-profile software-engineering or database conferences as our initial set of papers, while Lyu et al. did keyword search to get their initial set from not only software-engineering and database conferences but also security conferences, our collection could have fewer papers focusing on security antipatterns, and thus the smaller paper number. We do not communicate with Lyu et al. to get their publication list for comparison. However, given our focus on performance antipatterns, the exclusion of security-antipattern papers should have little to none impact on the number of publications with performance antipatterns collected from the literature; the impact is partially reflected by the small difference on paper numbers.

From the 47 papers and 1 book, 27 papers and the 1 book contain at least one performance antipattern. They are used to summarize known database-access performance antipatterns in the literature, and we refer to all of them as appropriate when presenting

Table 2.1: Subjects and bugs in our study

Subject	Abbreviation	Stars	Commits	Contributors	Language	Number of Bugs
BugZilla	BZ	323	9483	76	Perl	51
DNN	DN	712	15332	136	C#	26
Joomla!	JM	1969	96463	538	PHP	15
MediaWiki	MW	3514	32219	600	PHP	7
Moodle	MD	3011	97654	500	PHP	24
WordPress	WP	13866	41440	60	PHP	12
Odoo	OD	17524	133567	1041	Python	5
Sum						**140**

known antipatterns.

2.2.2 Bug Collection and Bug-Report Examination

Based on application popularity and bug-tracking system availability, we select seven open-source direct-accessing web applications to answer **RQ2** and **RQ3**: *BugZilla* [6], *DNN* [17], *Joomla!* [22], *MediaWiki* [41], *Moodle* [25], *WordPress* [42], and *Odoo* [31]. *BugZilla* is a bug-management system heavily used by Mozilla and other open-source projects. *DNN* is a popular content-management system, and its customers include large companies such as Bank of America, Canon, and BP. *Joomla!* and *MediaWiki* are also popular content-management systems, and they are used as subjects in a related study on database applications [134]. *Moodle* is a learning management system, which is widely used in schools, including North Carolina State University where several authors of this paper are affiliated with. *WordPress* is currently the most dominant content-management system on the market [39]. *Odoo* is a popular all-in-one business application. Table 2.1 shows the numbers of stars, commits, and contributors on GitHub. Our studied web applications are implemented using four different programming languages, including Perl, C#, PHP, and Python, and they all directly construct queries in application code without using ORM frameworks.

To collect performance bugs related to database accesses, we first search the bug-tracking systems of the selected web applications with keywords "performance," "timeout," and "slow" in order to retrieve performance bugs. After obtaining an initial set of performance bugs, we filter out bug reports that are not relevant to database accesses. Specifically, we keep only the reports whose description and comments contain database-related keywords, e.g., "database," "query," and "schema." Further, we include only bugs that have been closed with fixes. Following this process, our final performance-bug set contains 140 bug reports. In Table 2.1, the last column shows the number of bug reports that we collect for each application. In comparison, a previous study [166] focusing on performance

antipatterns in Ruby-on-Rails web applications studied 140 performance bugs reported in the bug-tracking systems of 12 applications and 64 performance issues identified through profiling. We focus on only performance bugs collected from bug-tracking systems, and our number of real-world performance bugs under study is comparable.

A bug report typically contains some bug description, followed by some discussions and comments on possible causes and fixes, and some intermediate fixes and the final committed fix. To answer **RQ2** and **RQ3**, our study centers around these preceding parts to understand the root causes and fix strategies of our collected bugs.

Every bug report is manually inspected and discussed by at least three authors to ensure the objectivity of our conclusions. We determine the root cause of each bug by examining each bug report to understand what particular reasons in program code, schemas, or database behaviors cause the performance bugs, and we determine the fix strategy of each bug by reviewing the patch submitter's description of the fix and inspecting the code in the patch to look for changes in the program code, queries, or schema.

For performance bugs whose root cause and fix strategy match known performance antipatterns, we label these performance bugs accordingly. For those without any matched known antipattern, we come up with new performance antipatterns to describe the root causes and fix strategies.

2.3 RQ1: Performance Antipatterns in the Literature

As discussed in Section 2.2.1, we identify 27 research papers and 1 book that discover or discuss at least one database-access performance antipattern through our literature search process. From these publications, we summarize 24 database-access performance antipatterns. Table 2.2 lists all the 24 performance antipatterns, and describes the root cause and fix strategy of each antipattern. All the 27 research papers and 1 book are cited under the "Origin" column when appropriate. Due to space limit, we do not provide examples for these antipatterns, and the readers can refer to the papers under the "Origin" column if more detailed explanations and examples are needed.

Although the 24 database-access performance antipatterns that we summarize are collectively covered by four papers in our initial paper set via the literature search [66, 118, 163, 166], none of them individually covers all of the 24 antipatterns. We also have to resolve some differences on categorization and reporting strategies, antipattern naming, and antipattern understanding to come up with the listed database-access performance antipatterns. In the remainder of this section, we discuss these issues related to the four papers that can collectively cover all the 24 antipatterns.

Table 2.2: Known database-access performance antipatterns summarized from the literature

ID	Name	Root cause	Fix strategy	Origins
AP-01	Inefficient queries	Issuing queries where semantically equivalent but more performant alternatives exist.	Using the more performant alternatives.	[53, 66, 99], [123, 166]
AP-02	Moving computation to the DBMS	Computing with the results of multiple queries, where the computation can also be done by the DBMS, and the network round-trip cost is larger than the query processing cost in the DBMS.	Moving the computation to the DBMS to save the network round-trip cost.	[66, 118, 163], [166]
AP-03	Moving computation to the server	Computing some results by the DBMS, which unfortunately is less performant compared with computing by the server, despite the increase in round-trip cost.	Moving the computation to the server despite extra round-trip cost.	[59, 166]
AP-04	Loop-invariant queries	Queries issued repeatedly in a loop always load the same database contents and hence are unnecessary.	Moving the query out of the loop and storing the queried results to intermediate objects.	[66, 166]
AP-05	Dead-store queries	The results of multiple queries are loaded to the same object, but the object is not used between some of these reloads.	Removing queries whose results are not used.	[66, 166]
AP-06	Queries with known results	Issuing queries whose results can be determined by examining the queries and program contexts without actually being executed.	Replacing the queries with the known results.	[66, 166]
AP-07	Redundant condition check	Queries issued inside condition checks and branches are identical and return the same results.	Storing the queried results to intermediate objects and using them in both the condition checks and branches.	[66]
AP-08	Not caching	Issuing multiple queries that are syntactically equivalent or of the same template without caching the query results.	Adding caching either using a new cache layer or storing the query results in static objects.	[69, 118, 147], [163, 176]
AP-09	Inefficient lazy loading	Issuing one query to retrieve N objects from one table, and N other queries to retrieve information related to the N objects from another table.	Issuing one query with a join clause of the two tables.	[66, 74, 75], [118, 120, 166]
AP-10	Not merging selection predicates	Issuing multiple `SELECT` queries where each loads only a subset of the needed rows.	Loading all needed rows in one query.	[52, 118, 120], [138]
AP-11	Not merging projection predicates	Issuing multiple `SELECT` queries where each loads only a subset of the needed columns.	Loading all needed columns in one query.	[52, 66, 118], [120]
AP-12	Inefficient eager loading	Eagerly loading associated objects that are too large.	Delaying the loading of the associated objects.	[66, 73, 166]
AP-13	Inefficient updating	Issuing N separate queries to update N database records.	Batching the N update queries into a single query.	[47, 66, 118], [119, 152, 166]
AP-14	Unnecessary column retrieval	Retrieving more columns than needed.	Retrieving only the columns that are needed.	[65, 66, 99], [118, 163, 166]
AP-15	Unnecessary row retrieval	Retrieving more rows than needed.	Only retrieving the rows that are needed.	[53, 65, 79], [99, 118]
AP-16	Unnecessary whole queries	The results of certain queries are completely unused.	Removing the queries.	[47, 68, 70], [71]

Table 2.2 (continued)

ID	Name	Root cause	Fix strategy	Origins
AP-17	Inefficient rendering	When a view file renders a set of objects, inefficient APIs are used.	Using more performant APIs for view rendering.	[166]
AP-18	Missing fields	Fields that are costly to be derived from other fields are not stored directly in database tables.	Storing the fields in database tables directly.	[166]
AP-19	Missing indexes	Appropriate indexes are not included in table schema.	Adding the necessary indexes.	[99, 123, 151], [166]
AP-20	Table denormalization	Issuing queries with fixed join predicates.	Storing the pre-joined, i.e., denormalized, table based on the fixed join predicates in the DBMS.	[163]
AP-21	Partial evaluation of projections	Issuing queries that mostly use a subset of stored fields in a table and the mostly unused fields are much larger in data size.	Partitioning the table column-wise into a table for frequently queried small fields and another for less queried large fields in the DBMS.	[163]
AP-22	Partial evaluation of selections	Issuing queries whose selection predicates contain constant values.	Storing table rows matching the predicates with constant values in the DBMS as a separate table.	[163]
AP-23	Unbounded queries	Queries returning an unbounded number of records to be displayed.	Pagination, i.e., splitting and displaying records on different pages.	[66, 118, 164], [163, 169, 170]
AP-24	Functionality trade-offs	Developers introducing new functionalities that are too costly.	Removing the costly new functionalities.	[164, 166, 169], [170]

2.3.1 Categorization and Reporting Strategies

Yang et al. [166] categorized the root causes of performance antipatterns into several high-level categories; this categorization is followed by Chen et al. [66]. In our 24 performance antipatterns, [AP-01] to [AP-03] can be categorized as inefficient computation, [AP-04] to [AP-08] can be categorized as unnecessary computation, [AP-09] to [AP-13] can be categorized as inefficient data accessing, [AP-14] to [AP-16] can be categorized as unnecessary data retrieval, [AP-17] is on its own, [AP-18] to [AP-22] can be categorized as database-design problems, and [AP-23] and [AP-24] can be categorized as application-design trade-offs. Under the context of ORM-based web applications, Yang et al. [166] considered [AP-01] to [AP-17] all as ORM API misuses at a higher level, but this categorization does not apply to direct-accessing applications, as they do not simply use APIs to access databases but have to construct queries in the applications.

Regarding [AP-01] to [AP-03], which are all categorized into the same high-level category, inefficient computation, while Chen et al. [66] reported this high-level category, they did not report the three performance antipatterns. Instead, they reported seven specific performance rules that are related to ORM APIs. Since these rules are not applicable to direct-accessing web applications, we report the three general performance antipatterns. For performance rules that are specific to languages or frameworks, we consider them as special cases of general performance antipatterns instantiated on specific applications, languages, or frameworks. Except the study by Chen et al. [66], the same strategy is followed by all the other three studies [118, 163, 166] among the four that collectively cover all the 24 antipatterns.

On the other hand, we also acknowledge that there is value in summarizing specific performance rules. The reason is that [AP-01] (inefficient queries) is very broad. For ORM-based web applications, many different ORM APIs can be misused and result in performance inefficiencies, and [AP-01] indeed can be further specialized based on different ORM-API misuses. Since we are interested in summarizing performance antipatterns that are applicable to both ORM-based web applications and direct-accessing web applications, we include only the general antipatterns.

2.3.2 Resolving Naming and Understanding Differences

The surveyed publications could sometimes use different names for the same database-access performance antipattern, and we choose the most intuitive name if this case occurs. Otherwise, we inherit the names without changing them. To this end, most of our antipattern names are inherited from the four study papers in our initial search set.

Specifically, from the study by Yang et al. [166], we inherit the names of [AP-01] to [AP-06], [AP-09], [AP-12], [AP-13], [AP-17] to [AP-19], and [AP-21]; from the study by Chen et al. [66], we inherit the names of [AP-07]; from the study by Lyu et al. [118], we inherit the names of [AP-08], [AP-10], [AP-11], [AP-14], [AP-15], [AP-20]; and from the study by Yan et al. [163], we inherit the names of [AP-22] to [AP-24].

Some database-access performance antipatterns have other names that are equally good as the ones that we choose and are worth mentioning. Specifically, [AP-09] (inefficient lazy loading) is also known as loop to join [118] or N+1 problems; [AP-13] (inefficient updating) is also known as unbatched writes [118]; and [AP-23] (unbounded queries) is also known as content display trade-offs [166].

We introduce the name of antipattern [AP-16] for cases that belong to unnecessary data retrieval but are not [AP-14] (unnecessary column retrieval) or [AP-15] (unnecessary row retrieval). While the study by Lyu et al. [118] included names for [AP-14] and [AP-15], it does not include an appropriate name for [AP-16].

Chen et al. [66] reported a new antipattern, mid-result misuse, which happens when there are multiple queries on the same object, and each query retrieves different columns. In our case, we consider it as a special case of [AP-11] not merging projection predicates. Since Chen et al. did not include the papers that originally discussed [AP-11] in their study, they reported mid-result misuse as a new antipattern.

2.4 RQ2: Coverage of Known Antipatterns on Database-Access Performance Bugs from Direct-Accessing Web Applications

Following the methodology described in Section 2.2.2, we collect 140 database-access performance bugs from seven direct-accessing web applications. After studying their root causes and fix strategies, we can match 107 of them with the 24 known database-access performance antipatterns described in Section 2.3.

Table 2.3 shows the numbers of database-access performance bugs matching known antipatterns in each of our selected applications. We also show the total numbers of performance bugs matching each known antipattern from the previous study focusing on ORM-based web applications using Ruby-on-Rails [166]. Since we focus on performance bugs collected from bug-tracking systems, we exclude the problematic actions that they identified through profiling. Thus, although antipattern [AP-17] was reported by Yang et al. [166], the corresponding number in the last column of Table 2.3 is 0, as this antipattern

Table 2.3: Numbers of performance bugs matching the 24 known antipatterns in applications selected by us and Yang et al. [166]

ID	BZ	DN	JM	MW	WP	MD	OD	**Sum**	[166]
AP-01	2	1	2	0	3	1	0	9	12
AP-02	2	2	1	0	0	0	1	6	4
AP-03	4	1	0	0	1	1	0	7	2
AP-04	0	0	0	0	0	1	0	1	5
AP-05	0	0	0	0	0	0	0	0	5
AP-06	0	1	1	0	0	0	0	2	7
AP-07	0	0	0	0	0	0	0	0	0
AP-08	5	2	1	0	0	2	0	10	0
AP-09	0	0	0	0	0	0	0	0	27
AP-10	16	0	0	0	0	0	0	16	0
AP-11	0	0	0	0	0	0	0	0	0
AP-12	0	0	0	0	0	0	0	0	1
AP-13	1	0	1	0	0	1	0	3	1
AP-14	3	0	1	0	0	0	0	4	4
AP-15	2	2	0	1	1	1	2	9	1
AP-16	3	0	1	0	3	2	0	9	3
AP-17	0	0	0	0	0	0	0	0	0
AP-18	1	2	0	0	0	0	0	3	5
AP-19	3	13	1	0	1	6	0	24	30
AP-20	0	0	0	0	0	0	0	0	0
AP-21	1	0	0	0	0	0	0	1	0
AP-22	0	0	0	0	0	0	0	0	0
AP-23	2	0	0	1	0	0	0	3	14
AP-24	0	0	0	0	0	0	0	0	19
Sum	45	24	9	2	9	15	3	107	140

does not appear in performance bugs collected from bug-tracking systems. Other antipatterns with their corresponding numbers as 0 in the last column were not reported by Yang et al. [166]. Antipatterns [AP-14] to [AP-16] were reported as a single antipattern, unnecessary data retrieval, by Yang et al. [166], and we further categorize their studied bugs falling into unnecessary data retrieval with a finer granularity.

In total, 107 of our studied bugs can be categorized with 15 out of the 24 known antipatterns, while performance bugs studied by Yang et al. [166] can be categorized with 16 out of the 24 known antipatterns. In this sense, these two studies are of similar representative on covering known database-access performance antipatterns. However, 33 out of our studied bugs cannot be categorized into known antipatterns, revealing that there

are new antipatterns not covered by the current research literature. This result shows that studying performance bugs in direct-accessing database-backed web applications indeed can improve the coverage of real-world database-access performance antipatterns.

Both studies cover [AP-01] to [AP-04], [AP-06], [AP-13] to [AP-16], [AP-18], [AP-19], and [AP-23]. The number of studied performance bugs matching [AP-19] (missing indexes) is the highest in both studies. This result shows that it is challenging for developers to pick the optimal indexes at the database design phase regardless of whether the web applications are direct-accessing or ORM-based. Among the 12 antipatterns covered by both studies, the number differences on [AP-15], [AP-16], and [AP-23] are larger than five. [AP-14] to [AP-16] are the three antipatterns falling into the same high-level category, unnecessary data retrieval, and our study has 14 more bugs matching [AP-14] to [AP-16] combined. On [AP-23] (unbounded queries), the study by Yang et al. [166] has 11 more bugs than ours. We find number differences on the other nine shared antipatterns not significant.

Seven antipatterns appear in the study by Yang et al. [166] or in our study but not both. Specifically, antipatterns appear in only our studied bugs are [AP-08] (not caching), [AP-10] (not merging project predicates), and [AP-21] (partial evaluation of projections), while antipatterns appear in only their studied bugs are [AP-05] (dead-store queries), [AP-09] (inefficient lazy loading), [AP-12] (inefficient eager loading), and [AP-24] (functionality trade-offs). Other than [AP-12] and [AP-21] where the number difference is one in both cases, the number differences for the other antipatterns are at least five. [AP-09] and [AP-10] are related, where both have N queries that can be merged into a single query, while [AP-09] further merges the one query, which returns N objects and leads to N queries, with the N queries into a single query. For the differences on [AP-05], [AP-08], and [AP-24], we find it difficult to come up with definite explanations.

Both studies do not have bugs matching five database-access performance antipatterns: [AP-07] (redundant condition check), [AP-11] (not merging projection predicates), [AP-17] (inefficient rendering), [AP-20] (table denormalization), and [AP-22] (partial evaluation of selection). Among these five antipatterns, the first two are generally applicable to both direct-accessing web applications and ORM-based web applications, although both studies do not have matching bugs; the third one was reported by Yang et al. [166] in problematic actions identified through profiling; and the last two are less likely to match performance bugs from bug-tracking systems than the first three as the last two involve high-level database-design changes.

Table 2.4: New antipatterns found in our studied performance bugs from direct-accessing web applications

ID	**Name**	BZ	DN	JM	MW	WP	MD	OD	**Sum**
AP-25	Existing indexes not leveraged	1	0	3	0	2	3	0	9
AP-26	Non-optimal force index	0	0	0	3	0	0	0	3
AP-27	Changing subqueries to join operations	0	1	0	0	0	4	1	6
AP-28	Changing join operations to subqueries	1	1	1	1	0	1	0	5
AP-29	Joining unused tables	1	0	1	0	0	1	1	4
AP-30	Unnecessary locks	1	0	0	1	0	0	0	2
AP-31	Subquery returning duplicated rows	1	0	0	0	0	0	0	1
AP-32	Conditions containing subsuming clauses	0	0	0	0	1	0	0	1
AP-33	Unnecessary `where` clause when all conditions are selected	1	0	0	0	0	0	0	1
AP-34	Unnecessary query construction	0	0	1	0	0	0	0	1
Sum		6	2	6	5	3	9	2	33

```
# Before fix
query = "SELECT comment_date_gmt FROM comments WHERE ... ORDER BY comment_date DESC LIMIT 1;"

# After fix
query = "SELECT comment_date_gmt FROM comments WHERE ... ORDER BY comment_date_gmt DESC LIMIT
    1;"

# Affected table schema
CREATE TABLE comment(
  ...
  (no KEY on comment_date)
  KEY comment_date_gmt (comment_date_gmt)
)
```

Figure 2.1: *WordPress* bug #4366. Before and after code snippets are shown.

```
# The affected query
query = "SELECT ... FROM '#__content' AS a LEFT JOIN ... LEFT JOIN '#__associations' AS asso
    ON asso.id = a.id ..."

- CREATE TABLE IF NOT EXISTS '#__associations'
-       ('id' varchar(50) NOT NULL, ...)
+ CREATE TABLE IF NOT EXISTS '#__associations'
+       ('id' INT(11) NOT NULL, ...)
```

Figure 2.2: *Joomla!* bug #29845. Patch diff results are shown.

2.5 New Performance Antipatterns

From the 33 bugs that do not match with known antipatterns found in the literature, we derive 10 new database-access performance antipatterns. Table 2.4 shows the overall results. Below, we first describe each new database-access performance antipattern in detail. For each antipattern, we first describe the root cause and fix strategy and then present real-world performance-bug examples. After that, we conclude this section with a discussion.

2.5.1 Details of the New Performance Antipatterns

[AP-25] Existing indexes not leveraged

Root cause: Due to unawareness of existing indexes or mismatches among queries and table schema definitions, the queries fail to leverage existing indexes.

Fix Strategy: Change the queries or table schema definitions so that existing indexes can be taken advantage of.

Examples: Figure 2.1 shows *WordPress* #4366, where the buggy code orders the results by the unindexed `comment_date`, and the fix is to order the results by `commend_data_gmt` that is already indexed.

Figure 2.2 shows *Joomla!* #29845, where the problem is more subtle. The problem is

```
# Before fix
query = "SELECT ... FROM page FORCE INDEX (page_random) WHERE page_namespace = $page AND
    page_is_redirect = '0' AND page_random >= 0 ORDER BY page_random LIMIT 1;"

# After fix
query = "SELECT ... FROM page WHERE page_namespace = $page AND page_is_redirect = '0' AND
    page_random >= 0 ORDER BY page_random LIMIT 1;"

# Affected table schema
CREATE TABLE page(
  ...
  KEY page_namespace (page_namespace)
  KEY page_random (page_random)
)
```

Figure 2.3: *MediaWiki* bug #59285. Before and after code snippets are shown.

```
$sql = "SELECT gi.id FROM {grade_items} gi
- WHERE ... AND gi.categoryid IN (
- SELECT gc.id FROM {grade_categories} gc
- WHERE gc.path LIKE ?)"
+ JOIN {grade_categories} gc ON gi.categoryid = gc.id
+ WHERE ... AND gi.courseid = ?
+ AND gc.path LIKE ?"
```

Figure 2.4: *Moodle* bug #42065. Patch diff results are shown.

mismatched column types between two tables: the join operation joins the `id` column in table `#__associations` with a column in another table, where the type of column `id` is string, but the type of the column in the other table is integer. When the database engine performs the join operation, it has to do an extra type conversion on the two columns with different data types. After casting, the index cannot be leveraged, causing significant performance slowdown. The patch changes the type of column `id` from `varchar` to `INT`.

[AP-26] Non-optimal force index

Root cause: Developers construct queries with a force index, which unfortunately is not optimal.

Fix Strategy: Removing the force index and using the optimal index.

Examples: Figure 2.3 shows *MediaWiki* #59285. With the force index on `page_random`, the query will check the where conditions starting with the indexed column `page_random` and then continue to check other conditions for each record. However, there are a lot of rows satisfying the condition `page_random >= 0`. So the size of rows to be checked with other conditions will still be large. The fix removes the force index on `page_random`. After patching, the database engine will start with checking the indexed column `page_namespace` and find a small number of rows satisfying the condition on `page_namespace`,

```
$sql = "SELECT c.*, ... FROM ... JOIN {course} c
- JOIN {event} e ON e.courseid = c.id
+ WHERE EXISTS (SELECT 1 FROM
+ {event} e WHERE e.courseid = c.id)
```

Figure 2.5: *Moodle* bug #32340. Patch diff results are shown.

leading to speedup on query execution.

[AP-27] Changing subqueries to join operations

Root cause: When a query can be implemented with subqueries or join operations, there are cases where using join operations is more efficient. This situation can happen when the size of the joined tables is not large.

Fix Strategy: Changing subqueries to join operations.

Examples: Figure 2.4 shows *Moodle* #42065, where the fix changes the way how the `categor-yid` field of the `grade_items` table is matched with the `id` field of the `grade_categories` table from an `IN` clause with a subquery to a `JOIN` operation. Since the size of each table is not large, the cost of joining those tables is smaller than the subquery, which will need to create and destroy temporary storage for subquery results.

[AP-28] Changing join operations to subqueries

Root cause: When a query can be implemented with subqueries or join operations, there are cases where using subqueries is more efficient. This situation can happen when the size of the joined tables is large.

Fix Strategy: Changing join operations to subqueries.

Examples: Figure 2.5 shows *Moodle* #32340. In order to fix this bug, the patch changes the way how the `id` field of the `course` table is matched with the `courseid` field of the `event` table from a `JOIN` operation to an `EXISTS` clause with a subquery. The fix improves performance as doing a subquery on the `event` table takes less time than joining multiple tables that are very large.

[AP-29] Joining unused tables

Root cause: Queries join tables that are not used.

Fix Strategy: Removing the tables being joined but not used by queries.

Examples: Figure 2.6 shows *BugZilla* #226284. For each value in the `@chfield` list, if it satisfies certain conditions, the value will be compared with the `actcheck.fieldid` field, where `actcheck` is an alias for the `bugs_activity` table. However, if no value in the

```
# Construct the join clause
- $join .= "LEFT JOIN bugs_activity actcheck");

# Construct the where conditions
...
my @list;
foreach my $f (@chfield) {
    if(...){
        push(@list, "actcheck.fieldid = " . get_field($f));
    } else {
        ...
    }
}

if(@list) {
+ $join .= "LEFT JOIN bugs_activity actcheck");
    foreach my $l (@list){
        $where .= $l;
    }
}

$where .= ...
...

$query .= $join . $where
```

Figure 2.6: *BugZilla* bug #226284. Patch diff results are shown.

`@chfield` list satisfies the conditions, it is unnecessary to join the `bugs_activity` table. The buggy code joins the `bugs_activity` table at the beginning, while the fix joins the table only if it will really be checked against with.

[AP-30] Unnecessary locks

Root cause: Tables get locked unnecessarily while executing some queries.

Fix Strategy: Remove the unnecessary locks in the queries.

Examples: Figure 2.7 shows *BugZilla* #301020, where the queries update some fields in the `components` table with a `WRITE` lock. Since the information of a component's `initialowner` and `initialqacontact` actually correspond to a user's username, which is stored in the `profiles` table. So, a `READ` lock needs to be added on the `profiles` table. No information stored in the `products` table is needed by these queries. Therefore, the `READ` lock on the `products` table is unnecessary and removed by the patch.

[AP-31] Subquery returning duplicated rows

Root cause: Subqueries in a query could return duplicated rows, leading to unnecessary computation in the query.

Fix Strategy: Removing duplicated rows from the results of subqueries.

Examples: Figure 2.8 shows *BugZilla* #818007, where the results of a `SELECT` subquery will

```
# Lock tables
- $dbh->bz_lock_tables('components WRITE', 'products READ', 'profiles READ');
+ $dbh->bz_lock_tables('components WRITE', 'profiles READ');

# Execute queries
$dbh->do("UPDATE components SET name = ? WHERE id = ?", $name, $component_id);
$dbh->do("UPDATE components SET initialowner = ? WHERE id = ?", $assignee_id, $component_id);
$dbh->do("UPDATE components SET description = ? WHERE id = ?", $description, $component_id);
$dbh->do("UPDATE components SET initialqacontact = ? WHERE id = ?", $qacontact_id,
    $component_id);

# Unlock tables
$dbh->bz_unlock_tables();
```

Figure 2.7: *BugZilla* bug #301020. Patch diff results are shown.

```
# Before fix
query = "SELECT ... FROM ... LEFT JOIN bugs LEFT JOIN (SELECT bug_id FROM comments WHERE ...)
    as c ON bugs.bug_id = c.bug_id ... WHERE ...;"

# After fix
query = "SELECT ... FROM ... LEFT JOIN bugs LEFT JOIN (SELECT DISTINCT bug_id FROM comments
    WHERE ...) as c ON bugs.bug_id = c.bug_id ... WHERE ...;"
```

Figure 2.8: *BugZilla* bug #818007. Before and after code snippets are shown.

be used as a table to be joined in the query. The problem is that the result of the subquery contains a large number of duplicates. After adding a `DISTINCT` keyword to remove the duplicates, the subquery dramatically reduces the amount of data returned, reducing the time needed for the query to join the result table of the subquery.

[AP-32] Conditions containing subsuming clauses

Root cause: A query could contain condition clauses where one may subsume another, leading to unnecessary computation in the query.

Fix Strategy: Removing the condition clauses that are subsumed by others.

Examples: Figure 2.9 shows *WordPress* #17152. The comparison with the whole `$search_term` string for string search in the text is not necessary, because if all substrings of `$search_term` have been searched in the text, there is no need to search the whole `$search_term` string in the text. The fix removes the condition clause that compares with the whole `$search_term` string; this clause is subsumed by condition clauses that compare with its substrings, and thus is unnecessary.

[AP-33] Unnecessary `where` *clause when all conditions are selected*

Root cause: A query could contain `where` condition clauses that compare a field with all its possible values, making all these clauses unnecessary.

```
foreach( (array) $q['search_terms'] as $term ) {
    $search .= "$searchand" . "(($wpdb->posts.post_title LIKE '%{$term}%') OR
        ($wpdb->posts.post_content LIKE '%{$term}%'))";
    $searchand = ' AND ';
}
- $search .= " OR ($wpdb->posts.post_title LIKE '%{$search_term}%') OR
     ($wpdb->posts.post_content LIKE '%{$search_term}%')";
```

Figure 2.9: *WordPress* bug #17152. Patch diff results are shown.

```
+ if ($params->param('bug_status')) {
+   my @bug_statuses = $params->param('bug_status');
+   if (scalar(@bug_statuses) == scalar(@::legal_bug_status)) {
+       $params->delete('bug_status');
+   }
+ }
+
+ if ($params->param('resolution')) {
+   my @resolution = $params->param('resolution');
+   if (scalar(@resolution) == scalar(@::legal_resolution)) {
+       $params->delete('resolution');
+   }
+ }
+

foreach my $field ($params->param()) {
    push(@where, join("OR" . @params->param($field)));
}
```

Figure 2.10: *BugZilla* bug #173571. Patch diff results are shown.

Fix Strategy: Removing the `where` condition clauses that cover all possible values.

Examples: Figure 2.10 shows *BugZilla* #173571. In this example, the query will build a clause into the `where` condition for each user-selected value. When all possible legal values of a column are selected, there is no need to include those clauses into the `where` condition. The fix removes the condition clauses on `bug_stats` and `resolution` to save the time for checking these conditions.

[AP-34] Unnecessary query construction

Root cause: A query is constructed but not sent to the DBMS for execution.

Fix Strategy: Remove the unnecessary query-construction code logic.

Examples: Figure 2.11 shows *Joomla!* #23164. Originally, the application first constructs a query, and then tries to load data from cache. If there is a cache hit, the query will not be executed. The fix instead gets cache results first (doing so is fast), and constructs the query only if there is a cache miss.

```
# Before fix
# Construct the query
$query .= "SELECT ...";
$query .= "FROM ...";
$query .= "LEFT JOIN ... ON ..."
$query .= "WHERE ..."

#Check if cache is empty
$modules = $cache->get($cacheid);
if (null === $modules){
    $modules = $db->loadObjectList();
}

# After fix
# Check if cache is empty
if(!($modules = $cache->get($cacheid)){
    # Construct the query
    $query .= "SELECT ...";
    $query .= "FROM ...";
    $query .= "LEFT JOIN ... ON ..."
    $query .= "WHERE ..."
    $modules = $db->loadObjectList();
}
```

Figure 2.11: *Joomla!* bug #23164. Before and after code snippets are shown.

2.5.2 Discussion

The 10 new database-access performance antipatterns unveiled by our study show the benefits of going beyond ORM-based web applications and studying performance bugs in direct-accessing web applications.

On the generality of the 10 new database-access performance antipatterns, we believe that all the 10 new antipatterns are applicable to other direct-accessing web applications. Although the numbers of some antipatterns are small, e.g., [AP-31] to [AP-34] each have only one matching performance bug in our study, the necessary program and query features are all general to all direct-accessing web applications, and similar mistakes can happen. Therefore, we consider these antipatterns as new ones despite the small numbers of studied bugs falling into some of the antipatterns.

Among the 10 new antipatterns, the first nine, i.e., [AP-25] to [AP-33], are also applicable to ORM-based web applications, as the underlying root causes are not specific to direct-accessing web applications and the necessary features involved in each antipattern are available in ORM frameworks. Since ORM-based web applications do not construct queries directly, [AP-34] (unnecessary query construction) is not directly applicable to ORM-based web applications. However, an analogy exists, i.e., ORM frameworks could perform unnecessary query construction while translating ORM-API calls to queries.

Among the 33 performance bugs matching new performance antipatterns, 12 of them match antipatterns [AP-25] or [AP-26], both of which are related to database indexes. We

have discussed a similar finding in Section 2.3 that the number of performance bugs matching [AP-19] missing indexes is the largest both in our study and the study by Yang et al. [166]. Our results on direct-accessing web applications show that the challenges of using database indexes effectively go beyond the phase of database table design but also lie in how to select and use appropriate indexes.

Similar to the known performance antipatterns, where duos of opposite antipatterns exist, e.g., [AP-02] (moving computation to the DMBS) vs. [AP-03] (moving computation to the server), and [AP-09] (inefficient lazy loading) vs. [AP-12] (inefficient eager loading), one duo of opposite exists in our new antipatterns, i.e., [AP-27] (changing subqueries to join operations) vs. [AP-28] (changing join operations to subqueries). This case serves as another example that one-fits-all design does not exist for accessing database efficiently in web applications.

Although only one antipattern can be considered as unique for direct-accessing web applications, i.e., [AP-34] (unnecessary query construction), detecting performance bugs in direct-accessing web applications using antipatterns shared with ORM-based web applications may encounter extra challenges due to the difference between ORM-based web applications and direct-accessing web applications. For example, it will be necessary to implement string analysis [76] or other necessary techniques to handle dynamically generated queries in direct-accessing web applications. We leave for future work the design, development, and evaluation of bug detection techniques that leverage the 34 database-access performance antipatterns.

2.6 Threats to Validity

The validity of our study results may be subject to multiple threats. Below we describe potential threats and our ways to address them.

The first threat is the likely incompleteness of our surveyed publications. We alleviate this threat by following the state-of-the-art literature-survey methodology, and we cross-check our results with the previous literature surveys on the surveyed publications and the reported antipatterns.

The second threat is the likely lack of representativeness of the studied applications. To minimize this threat, we choose popular open-source applications with a significant user base. Although *BugZilla* has the largest number of studied bugs, it does not bias our results, because the overall distribution of categorized bugs spreads across different web applications and antipatterns. So the characteristics of our studied bugs can likely be generalized to other database-backed web applications.

The third threat is that we may miss relevant bug reports during our search for performance bugs. We mitigate this threat by using keyword search together with bug categories and tags. We also search bug descriptions and comments in addition to bug report summaries, as developers tend to use common terms in the description and comments.

The fourth threat is related to our manual inspection of the collected bug reports. The manual inspection is independently performed and verified by at least three authors to alleviate this threat. If there are different opinions on a bug report, we discuss the bug report together to reach a consensus.

2.7 Conclusion

In this chapter, we have presented a comprehensive empirical study that characterizes performance antipatterns related to database accesses in web applications. From our literature survey, we have summarized and reported a total of 24 known performance antipatterns, and the comprehensiveness of our results makes it a great reference for future work on database-access performance antipatterns. Based on real-world performance bugs from direct-accessing web applications, we have found 10 new database-access performance antipatterns that are not previously reported in the research literature. Our study results can guide future research in combating performance bugs related to database accesses in web applications.

CHAPTER

3

A CHARACTERISTIC STUDY OF DEADLOCKS IN DATABASE-BACKED WEB APPLICATIONS

3.1 Introduction

In web applications, the core business logic is a group of *request handlers*, which are responsible for handling incoming HTTP requests. Depending on the number of requests involved, deadlocks can be categorized as *inter-request deadlocks* where the deadlocks happen between request handlers for two or more requests, *intra-request deadlocks* where the deadlocks happen within a request handler while handling one request, and *non-request deadlocks* where the deadlocks happen without involving request handling but in other execution phases of the web applications, e.g., when the applications start, shutdown, restart, or perform background tasks.

Web-application deadlocks could involve different types of resources. As web applications are commonly backed by databases on the server-side, database locks could be one important type of resources involved in web-application deadlocks. Language-level synchronization objects can also be involved, depending on the support for concurrency

and synchronization provided by different web-application development languages. For example, Java has more mature support for multithreading compared with PHP and Python. Lastly, as different paradigms and frameworks for developing web applications, e.g., Object Relational Mapping (ORM) and event-driven Node.js, are being proposed and adopted, synchronization objects in these frameworks and libraries can also be involved. Among these lock types, database locks are unique in that SQL queries could lead to *implicit* lock acquisition due to database internals.

Most existing work on deadlocks focus on multi-threaded programs, including characteristic studies [84, 115] and various techniques for detection [18, 20, 45, 46, 61, 62, 77, 81, 82, 96, 97, 107, 133, 144, 177], avoidance [77, 158, 159], prevention [98, 177], testing [143], and fixing [60, 108]. Since these general techniques focus on modeling language-level locks, they will not be able to handle deadlocks on database locks that are not explicitly requested in application code. For web-application deadlocks not related to concurrent request handlers or database locks, it is also not clear how helpful existing techniques are.

Specific to web-application deadlocks, existing techniques all focus on database-lock deadlocks, and detect-and-recover is the most well-known approach. Specifically, major databases, e.g., MySQL, PostgreSQL, and SQL Server, provide deadlock detection capability[12, 15, 27]. Upon a detected deadlock, a victim will be chosen, and the web application could retry the victim transaction. Databases also provide error logs with which application developers can diagnose the deadlocks and fix the root cause of these deadlocks if they choose to.

However, deadlocks on database locks are difficult to understand even with database logs. For example, someone posted the following question on StackOverflow upon seeing error logs about a deadlock from MySQL/InnoDB [40].

> "Why MySQL starts deadlocking when this simple command of scheduling a job is executed concurrently?
> If it is really true that InnoDB is expected to create deadlocks even in normal circumstances, then how is MySQL expected to be used in any production database which is expected to have more concurrent users? Am I missing something?"

Since the aforementioned StackOverflow question has no accepted answer yet, we use a deadlock example from the MySQL manual [1] shown in Listing 3.1 to illustrate the challenges of deadlock understanding. In Listing 3.1, three transactions try to insert the same value on the primary key in sequence, and then the first transaction rolls back, after which, the second and third transactions will be in a deadlock.

To fully understand how this sequence of queries leads to the deadlock, one needs to

```
CREATE TABLE t1 (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;

START TRANSACTION; /* TX1 */
INSERT INTO t1 VALUES(1);
                   START TRANSACTION; /* TX2 */
                   INSERT INTO t1 VALUES(1);
                                      START TRANSACTION; /* TX3 */
                                      INSERT INTO t1 VALUES(1);
ROLLBACK;
```

Listing 3.1: An example from MySQL's official manual

know the locking strategy followed by the underlying database storage engine and different locks requested by different queries being executed. Note that sometimes multiple locks could be requested during different phases of executing one query. While some manuals for the locking strategy used by database storage engines are usually provided by vendors, they do not seem to be enough to help application developers quickly understand web-application deadlocks on database locks, as exemplified by the aforementioned StackOverflow question. Facing these challenges, application developers could benefit from a characteristic study of real-world deadlocks on database locks, with which they can learn common patterns and acquire the necessary knowledge on database locking useful for deadlock understanding.

Beyond the detect-and-recover approach with support primarily from the database community, the software-engineering community has also contributed to the testing of deadlocks in database-backed applications [86] and prevention of deadlocks on database locks [87]. However, existing techniques only model the locking behavior in database queries very conservatively, and the example in Listing 3.1 is beyond the capability of these techniques. It is unclear how well existing techniques can cover real-world deadlocks on database locks.

To complement the current state of the art, in this work, we conduct a characteristic study of real-world deadlocks from database-backed web applications. We start our study with the following research question:

- **RQ1:** What are the common types of deadlocks in web applications regarding the number of HTTP requests and deadlock resources, and how these characteristics are impacted by application differences?

To answer **RQ1**, we do keyword search in the bug-tracking systems of 106 database-backed web applications, covering applications developed with major paradigms and languages, and find 49 real-world deadlocks in web applications. We characterize these 49 deadlocks based on the number of HTTP requests and deadlock resources involved in them. Our results suggest that inter-request deadlocks on database locks are not only the

most common but also the most challenging type of deadlocks in web applications, which is worth further investigation. As our keyword search only returns deadlocks in a subset of web applications, we also study the relationship between application characteristics and the number of deadlocks, and our results suggest that both development paradigm and project history could affect the number of deadlocks.

We proceed with the following two research questions to further study web-application deadlocks on database locks:

- **RQ2:** What are the common types of web-application deadlocks on database locks?
- **RQ3:** What are the common fixing strategies of web-application deadlocks on database locks?

To answer **RQ2** and **RQ3**, we use the 36 deadlocks on database locks that we collect while answering **RQ1**, and we further complement the bug set with 27 deadlocks based on StackOverflow questions. We characterize these 63 deadlocks into four different hold-and-wait cycles, depending on the complexity of resources involved. To make our study results useful for developers to understand database-lock deadlocks they may encounter, we further divide three out of the four types of cycles into 12 patterns and provide an example for each pattern. For each example, we describe the queries and the locks requested by these queries in detail. Among all the different categories of database-lock deadlocks, existing work [86, 87] may only be able to handle one pattern that is the most straightforward. Compared with the patterns, we find fixing strategies more straightforward to understand, and we also summarize our findings.

Overall, we expect our results can (1) ease the task of deadlock understanding for application developers and (2) guide tool researchers and developers to design and implement comprehensive tool support for deadlocks in web applications.

3.2 Methodology

In this section, we first describe our methodology on how we collect and analyze bug reports related to deadlocks from real-world web applications developed using different programming paradigms and frameworks, and we then describe our methodology on how we collect and analyze StackOverflow posts related to deadlocks on database locks. To answer **RQ1**, we use deadlock reports from real-world web applications. To answer **RQ2** and **RQ3**, we use real-world web-application deadlocks on database locks labeled after answering **RQ1** together with StackOverflow questions.

Our study includes three types of web applications, (1) classical ones that access databases by constructing SQL queries directly, (2) those implemented on top of ORM frameworks, and (3) those implemented on top of the Node.js framework. For classical web applications, we start with the application list from the study of performance antipatterns in classical web applications [150]. For ORM web applications, we start with the application list from the study of concurrency control in ORM web applications [55]. For Node.js applications, we start with the application list from the concurrency-bug study for Node.js applications [156] but exclude those that are just libraries but not complete applications. We also include other open-source web applications that we are aware of and those we run into during our study, e.g., some StackOverflow questions mention the names of web applications we originally do not include. To this end, our application set includes 11 classical, 77 ORM, and 18 Node.js web applications.

We follow the methodology taken by existing studies on concurrency bugs in multi-threaded applications [115], performance bugs in web applications [150, 163, 166], and non-deadlock concurrency bugs in web applications [78, 136, 156] while collecting and studying bug reports related to deadlocks.

To collect bugs related to deadlocks, we first search for closed bug reports in each application's issue-tracking system with the keyword "deadlock(s)." We do not include other keywords in our search because we would like to study bugs that are determined by application developers as deadlocks, under which case we believe the well-known word "deadlock(s)" will appear in the bug report. After keyword search, we obtain a total of 546 bug reports, i.e., 384 reports from 10 classical web applications, 148 reports from 22 ORM web applications, and 14 reports from 7 Node.js web applications.

With this initial set of bug reports, we filter out ones that only mention the word "deadlock(s)" but are actually not deadlocks. For example, sometimes application developers may call a hang bug due to infinite loops as deadlock. We also filter out bug reports that do not contain sufficient information for us to understand. A bug report typically contains some bug description, followed by some discussion and comments on possible causes and fixes, some intermediate fixes, and the final committed fix. Every bug report is manually inspected and discussed by at least two authors to ensure the objectivity of our conclusions. We determine the root cause of each bug by examining each bug report to understand what particular reasons in program code, schemas, or database behaviors cause the deadlock bugs, and we determine the fix strategy of each bug by inspecting its accepted patch for changes in program code, queries, or schema and reviewing the patch submitter's description of the fix.

Following this process, our final set has 49 closed reports with sufficient information

Table 3.1: Web applications and numbers of bugs being studied and their overall characteristics

Programming Paradigm	**Application (Abbreviation)**	**Server-Side Development Language**	Non-Request		Intra-Request		Inter-Request		
			Thread Sync. (Lock Only)	**Database Lock**	**Thread Lock**	**Database Lock**	**Thread Lock**	**Database Lock**	**Cache Lock**
Classical	MediaWiki (MW)	PHP	-	-	-	2	-	16	1
	Odoo (OD)	Python	-	-	-	4	-	-	-
	Drupal (DPL)	PHP	-	-	-	-	-	3	-
	Sonar (SNR)	Java	1 (1)	-	-	-	-	2	-
	BugZilla (BZ)	Perl	-	-	-	-	-	1	-
	OpenMRS (MRS)	Java	3 (2)	-	-	-	-	1	-
	Gerrit (GRT)	Java	3 (2)	-	-	-	2	-	-
ORM	Gitlab (GL)	Ruby on Rails	-	-	-	-	-	1	-
	Discourse (DC)	Ruby on Rails	-	-	-	-	-	1	-
	Spree (SPR)	Ruby on Rails	-	-	-	-	-	1	-
	Openstreetmap (OSM)	Ruby on Rails	-	-	-	-	-	1	-
	Lobsters (LOB)	Ruby on Rails	-	-	-	-	-	1	-
	AWX (AWX)	Django / Python	-	-	1	-	-	-	-
	Sentry (SEN)	Django / Python	-	-	2	-	-	-	-
Node.js	Ghost (GHO)	Javascript	-	1	-	-	-	1	-

Table 3.2: Accumulated numbers of deadlocks involving different numbers of requests and different types of resources

	Thread Sync (Lock Only)	Database Lock	Cache Lock	Total
Non-Request	7 (5)	1	0	8
Intra-Request	3 (3)	6	0	9
Inter-Request	2 (2)	29	1	32
Total	12	36	1	49

for us to determine that their root causes are deadlocks. In comparison, the study of concurrency bugs in multi-threaded applications includes 31 deadlocks [115]. Table 3.1 shows the names and the numbers of deadlocks for each application. Note that the previous concurrency-bug study on Node.js applications and libraries states that they found no deadlock [156]. For the two Node.js deadlocks we find, one of them is reported after the study is published. The other is reported before the study is published, but the deadlock happens during the application start phase after a database upgrade, which could be the reason why it was not included by the authors of the previous study [156].

To further complement our understanding of deadlock patterns, we also search questions on StackOverflow for analysis. We use 35 different combinations of tags and keywords, e.g., "deadlock," "database," "MySQL," and "web application," for question search. For searches returning more than 50 questions, we include the first 50 with the highest votes. Otherwise, all returned questions are included. To this end, we obtain an initial set of 81 unique questions. We then manually filter out questions without sufficient information for us to understand, e.g., questions with no answer or no discussion. Following this process, we finally obtain a set of 27 questions.

For each bug report and StackOverflow question, two authors first independently examine all available information, including description, discussion, database log, source code, and fixes to make their own conclusion. Then, the two inspectors cross-check with each other with more authors involved in the discussion to reach a final conclusion.

3.3 RQ1: Overall Deadlock Characteristics

In this section, we first discuss the overall characteristics of the deadlocks we collect, and we then discuss how application differences affect these characteristics.

3.3.1 Overall Characteristics of Collected Deadlocks

We first categorize web-application deadlocks based on the number of HTTP requests and the types of resources involved in deadlocks. On request numbers, we categorize them into non-request, intra-request, and inter-request deadlocks, which need zero, one, and more than one HTTP request, respectively. On resource types, we differentiate database locks, thread synchronization that includes locks and condition variables, and other locks explicit in application code, e.g., cache locks. Table 3.1 shows the numbers of deadlocks involving different numbers of requests and different types of resources for each application, and Table 3.2 shows the accumulated numbers.

Among the 8 non-request deadlocks, 7 of them are on thread locks in applications developed with Java. These deadlocks either happen during the starting, shutdown, or restarting phase, or they are triggered while performing offline or background tasks. In these scenarios, deadlocks happen due to concurrency internal to the language but not due to external HTTP requests that arrive concurrently. Thus, it is not surprising that these 7 deadlocks are in applications developed with Java. 5 of them only involve locks. Among them, 4 are fixed by removing unnecessary locks, and the remaining 1 is fixed by changing application logic to make the two deadlock parties not concurrent; The other 2 involve condition variables, and they are fixed by adding the missing signal or removing the untimely wait. Overall, they are similar to classical thread deadlocks in Java. The remaining non-request deadlock is in a Node.js web application. The deadlock happens when the web application starts after a database upgrade, and it involves concurrent `UPDATE` queries. To fix this deadlock, programmers choose to issue these queries sequentially.

Among the 9 intra-request deadlocks, 3 of them are on thread locks in applications developed with Django, which is a Python-based web framework with ORM support. These deadlocks are all due to recursive lock operations on the same lock, and they are fixed by either using a reentrant lock instead of a normal lock or removing unnecessary calls that try to acquire the same lock. In the remaining 6 intra-request deadlocks on database locks, 4 happen in *Odoo*, which uses PostgreSQL as its backend database, and 2 happen in *MediaWiki*, which involves asynchronous execution, and we will discuss their deadlock patterns in Section 3.4.

Among the 32 inter-request deadlocks, 3 of them are not on database locks, where 2 are on thread locks and 1 is on a cache lock. They happen all due to missing `unlock` calls while handling one HTTP request, and they are fixed by adding the missing `unlock` calls. The remaining 29 are all on database locks, and we will detail them in Section 3.4.

From the discussion above, we can see that existing techniques can handle the studied deadlocks on thread synchronization or cache lock, regardless of the request number.

Table 3.3: Patterns of database-lock deadlocks and their numbers

Pattern	Nested TXes	Simple Cycles			Cycles with a Lock Held by Multi TXes			Cycles with Lock Queues						Total
		Pattern-1	Pattern-2	Pattern-3	Pattern-4	Pattern-5	Pattern-6	Pattern-7	Pattern-8	Pattern-9	Pattern-10	Pattern-11	Pattern-12	
# in App	4	6	0	0	2	4	7	2	2	3	0	3	3	36
# in SO	0	2	6	1	5	1	8	0	2	0	1	0	1	27

From the accumulated numbers in Table 3.2, we can see that inter-request deadlocks are more common than non-request and intra-request deadlocks, which is likely due to the nature of web applications that their core logic is handling concurrent requests. We can also see that deadlocks on database locks are more common, which is likely due to the deep coupling between web applications and databases. To this end, inter-request deadlocks on database locks are the most common, and they are also the most challenging for existing techniques to handle due to two challenges, i.e., they require new techniques to (1) analyze the relationship between different requests and (2) model database locking behavior. To handle non-request and intra-request deadlocks on database locks, while they would not exhibit the first challenge, we still need to handle the second challenge.

For the relationship between request numbers and deadlock resources, we can see that both thread locks and database locks could be involved in non-request, intra-request, and inter-request deadlocks. Therefore, they are two orthogonal dimensions for web-application deadlocks.

3.3.2 Application Differences vs. Deadlock Characteristics

For the relationship between deadlock resources and development languages, deadlocks on thread synchronization are more common in web applications developed with languages that provide mature support for concurrency and synchronization, i.e., Java in our case, but deadlocks on threaded synchronization can also happen in applications developed with other languages, as more languages have now gradually added support for concurrency and synchronization.

For the relationship between development paradigms and numbers of deadlocks, we can see classical applications have more deadlocks compared with web applications based on ORM frameworks or Node.js. Note that we also searched many applications with results of zero deadlocks, as described in Section 3.2. This result could be due to two reasons. First, classical web applications generally have a longer development history. Secondly, ORM web-application developers reportedly prefer not to use transactions in their code [55], which is a necessary condition for database-lock deadlocks to happen.

3.4 RQ2: Patterns of Database-Lock Deadlocks

Following the process discussed in Section 3.2, we identify 36 deadlock bugs on database locks from real-world web applications and 27 such deadlocks from StackOverflow questions. As database-lock deadlocks happen between concurrent transactions, but the source

of concurrency does not affect the patterns for database-lock deadlocks much, we include all non-request, intra-request, and inter-request cases in this section.

From these cases, we summarize four patterns of deadlocks on database locks that differ on the types of resources involved in deadlock hold-and-wait cycles, and Table 3.3 shows the overall results. Specifically, in the order of increasing complexity, the four cycle patterns are: (1) *Nested Transactions*, where a program creates two database connections in one thread, starts a transaction in each connection, and requests two conflicting locks, and this is similar to deadlocks caused by nested lock acquisition in multi-threaded programs; (2) *Simple Cycles* that involve locks on two rows; (3) *Cycles with a Lock Held by Multiple Transactions*, which involve locks that can be held by multiple transactions simultaneously, and they require extra modeling efforts; and (4) *Cycles with Lock Queues* that further involve lock queues, which is due to how locks are implemented internally in databases.

In this section, we first provide necessary background concepts on database locking, and we then detail the four deadlock cycle patterns with more subpatterns and concrete examples. As our goal is to help application developers understand database-lock deadlocks that they may encounter in the future, our subpatterns and examples are very detailed. We do not try to exhaustively enumerate all possible patterns that may occur in theory, but we categorize and show real-world cases that we see in web applications and StackOverflow questions.

3.4.1 Background on Locking Strategy

All database-lock deadlocks that we study are either on MySQL/InnoDB or PostgreSQL. Both MySQL/InnoDB and PostgreSQL use multiversion concurrency control (MVCC) and provide four isolation levels following SQL standard, i.e., Read Uncommitted, Read Committed, Repeatable Read, and Serializable, but their locking strategies are different. In the deadlocks that we study, 32 application and 22 StackOverflow deadlocks are on MySQL/InnoDB, and 4 application and 5 StackOverflow deadlocks are on PostgreSQL.

To understand the deadlocks on PostgreSQL locks, only general knowledge of standard SQL is needed, e.g., the concepts of clustered index, secondary index, primary key, and non-primary index. Such knowledge is assumed in this section. Next, we describe concepts that are fundamental for understanding deadlocks on MySQL/InnoDB locks. Due to space limitations, we are not trying to be comprehensive in this subsection, but we focus on two concepts, i.e., lock modes and lock types. Later in this section, we will describe the mode and type of locks being requested by each query in our examples.

In MySQL/InnoDB, locks can be in two modes: (1) a *shared* (S) lock permits the trans-

action that holds the lock to read some rows, and (2) an *exclusive* (X) lock permits the transaction to modify some rows. Locks can be in one of four types: (1) *Record Lock*, which is a lock on an index record, (2) *Gap Lock*, which is a lock on a gap between index records, or a lock on the gap before the first or after the last index record, (3) *Next-Key Lock*, which is a combination of a record lock on the index record and a gap lock on the gap before the index record, and (4) *Insert-Intention Lock*, which is a type of gap lock set by `INSERT` operations prior to row insertion.

Under MVCC, locks are requested automatically for SQL queries based on the isolation level, and queries could be blocked if the requested locks conflict with locks granted to other transactions. Unless otherwise specified, the isolation level in our studied bugs is repeatable read. All locks are released when a transaction is committed or aborted. Transactions can be started and committed explicitly, or a query that is not in any transaction is a transaction by itself.

3.4.2 Cycles with Nested Transactions

The 4 intra-request deadlocks in PostgreSQL-backed *Odoo* are due to nested transactions in one execution thread, where one request handler first makes a database connection, starts one transaction, and requests one lock, and it then makes a new database connection within the same execution thread, starts a new transaction, and requests a conflicting lock.

3.4.3 Simple Cycles

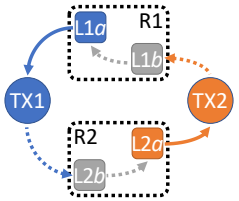

Figure 3.1: A simple cycle

Figure 3.1 shows the simple deadlock cycle with locks on two records `R1` and `R2`. In the diagram, transaction `TX1` holds lock `L1a` and waits for lock `L2b`, and transaction `TX2` holds lock `L2a` and waits for lock `L1b`. Further, locks `L1a` and `L1b` are conflicting, and locks `L2a` and `L2b` are conflicting. Note that `L1a` and `L1b` can be one lock, and `L2a` and `L2b` can also be one lock. Depending on how many SQL queries are involved, we further divide deadlocks

```
CREATE TABLE live_measures(
   UUID VARCHAR(40) NOT NULL, ...
);
ALTER TABLE live_measures ADD CONSTRAINT PK_LIVE_MEASURES PRIMARY KEY(UUID);

START TRANSACTION; /* TX1 */
UPDATE live_measures SET ... WHERE UUID=2;
                  START TRANSACTION; /* TX2 */
                  DELETE FROM live_measures WHERE UUID=1;
UPDATE live_measures SET ... WHERE UUID=1;
                  DELETE FROM live_measures WHERE UUID=2;
```

Listing 3.2: *Sonar* #11097

```
CREATE TABLE problem_table(
   id INT(11) NOT NULL,
   type enum('TYPE1','TYPE2','TYPE3') NOT NULL,
   source VARCHAR(16) DEFAULT NULL,
   PRIMARY KEY (id),
   KEY type_idx (type), ...
);

START TRANSACTION; /* TX1 */
UPDATE problem_table SET ... WHERE id=2;
                  START TRANSACTION; /* TX2 */
                  INSERT INTO temp SELECT ... FROM problem_table p WHERE p.type IN ('TYPE1',
                      'TYPE2') AND p.source='FOO';
UPDATE problem_table SET ... WHERE id=1;
```

Listing 3.3: StackOverflow #40653848

with simple cycles on database locks into three categories with four, three, and two queries, respectively.

[Pattern-1] Simple Cycles with Four Queries

Description: Pattern-1 deadlocks involve four queries from two transactions, with two queries from each transaction, and these queries access the database with primary-key values or unique-index values specified.

Example: Listing 3.2 shows the deadlock in *Sonar* #11097 [7]. The four queries involved in the deadlock either `UPDATE` or `DELETE` one row with values of the primary key specified. Thus, they all acquire a record lock for its corresponding row in the exclusive mode, but the two transactions acquire the two locks in the opposite order, resulting in a deadlock.

[Pattern-2] Simple Cycles with Three Queries

Description: Pattern-2 deadlocks involve three queries from two transactions, with two queries in one transaction and one query in the other transaction. The one query could request multiple locks due to several different reasons: full table scan during query execution, multiple tables being involved, or multiple indexes being involved.

Examples: Listings 3.3 and 3.4 show two examples where one query leads to a full table scan

```
CREATE TABLE jobs(
   jid INT(11) NOT NULL,
   status VARCHAR NOT NULL, ...
   PRIMARY KEY (jid)
);

START TRANSACTION; /* TX1 */
UPDATE jobs SET ... WHERE jid=2;
                   START TRANSACTION; /* TX2 */
                   SELECT ... FROM jobs WHERE status='new' FOR UPDATE;
UPDATE jobs SET ... WHERE jid=1;
```

Listing 3.4: StackOverflow #1851528

and locks multiple primary-key records, and they are based on StackOverflow questions #40653848 [21] and #1851528 [28], respectively. In Listing 3.3, the SELECT subquery of INSERT in TX2 will perform a full table scan and acquire a shared record lock on each primary-key record that satisfies the WHERE condition. Although there is an index on type, the database engine still decides to perform

```
START TRANSACTION; /* TX1 */
INSERT INTO phppos_sales VALUES (...);
                   START TRANSACTION; /* TX2 */
                   CREATE temporary TABLE temp SELECT ... FROM phppos_sales_items INNER JOIN
                       phppos_sales ON ... INNER JOIN ... WHERE ...;
INSERT INTO phppos_sales_items VALUES (...);
```

Listing 3.5: StackOverflow #23768456

a full table scan. In Listing 3.4, the SELECT FOR UPDATE query in TX2 will perform a full table scan as well and acquire an exclusive record lock on each primary-key record that satisfies the WHERE condition. The database engine performs a full table scan in this case, as the field in the WHERE condition is not indexed. In both cases, the two queries from TX1 request exclusive record locks on two rows but in an order opposite with the order that the query from TX2 locks the same two rows during the full table scan.

Listing 3.5 shows an example based on StackOverflow question #23768456 [26], and it is one example where one query locks rows from two different tables due to joined tables. In TX2, the SELECT subquery of CREATE is performed on a table joined from two existing tables. For each row matching the WHERE condition, the corresponding row in table phppos_sales_items will be locked first, and then the corresponding row in table phppos_sales will be locked. On the other hand, the two queries in TX1 request exclusive record locks on the two rows of these two tables in a different order, resulting in a deadlock.

Listing 3.6 shows an example based on StackOverflow question #2560070 [29], where one query locks rows from two tables due to foreign keys. In TX1, the SELECT FOR UPDATE query requests an exclusive lock on the row with id=1000 in table A. Then, the INSERT query in TX2 first gets an exclusive record lock on the row with id=1 just being inserted in table

```
create table A (id INT(11) PRIMARY KEY);
create table B (
   id INT(11) PRIMARY KEY,
   aid INT(11), ...
   FOREIGN KEY (aid) REFERENCES A(id)
);

START TRANSACTION; /* TX1 */
SELECT * FROM A WHERE id=1000 FOR UPDATE;
                    START TRANSACTION; /* TX2 */
                    INSERT INTO B (id, aid, ...)
                         VALUES (1, 1000, ...);
INSERT INTO B (id, aid, ...)
     VALUES (1, 1000, ...);
```

Listing 3.6: StackOverflow #2560070

```
CREATE TABLE tab1 (
   id INT(11) NOT NULL AUTO_INCREMENT,
   sn VARCHAR(20) NOT NULL,
   is_fetch TINYINT(1) NOT NULL DEFAULT '0' , ...
   PRIMARY KEY (id),
   KEY sn (sn),
   KEY is_fetch (is_fetch),
);

START TRANSACTION; /* TX1 */
SELECT sn FROM tab1 WHERE is_fetch=0
     LIMIT 200 FOR UPDATE;
                    START TRANSACTION; /* TX2 */
                    INSERT IGNORE INTO tab1 (sn, is_fetch, ...)
                         VALUES ('4287', 0, ...);
UPDATE tab1 SET is_fetch=1
    WHERE sn in ('4287', ...);
```

Listing 3.7: StackOverflow #24327317

B, and it will then request a shared record lock on the row with id=1000 in table A, as the primary key of table A is a foreign key in table B. However, this request from TX2 is blocked due to the lock on that row held by TX1. Finally, the INSERT query in TX1 will also try to insert into table B, but it gets blocked during duplicate-key checking by TX2, as a row satisfying id=1 has been inserted into table B by TX2 already.

Listing 3.7 shows an example based on StackOverflow question #24327317 [4], where one query locks rows in two indexes. In TX1, the SELECT FOR UPDATE query acquires an exclusive next-key lock on every record in index is_fetch satisfying is_fetch=0 and a gap lock on the range after the last record satisfying is_fetch=0. These ranges are locked to prevent other transactions from inserting records satisfying is_fetch=0 in the is_fetch index concurrently. Then, the INSERT query in TX2 inserts a row whose is_fetch field equals 0. It successfully inserts the record to the primary index and acquires an exclusive lock on the newly inserted primary-index record, but it gets blocked while requesting an exclusive insert-intention lock on secondary index is_fetch, as it falls into the range after last is_fetch=0 record, which has been locked by TX1. Finally, the database engine chooses

```
CREATE TABLE fruit_setting (
   id BIGINT(20) NOT NULL AUTO_INCREMENT,
   aid VARCHAR(32) NOT NULL,
   eid VARCHAR(32) NOT NULL,
   mykey VARCHAR(32) NOT NULL, ...
   PRIMARY KEY (id),
   KEY i_aid_mykey (aid, mykey),
   UNIQUE KEY i_eid_mykey (eid, mykey), ...
);
INSERT INTO fruit_setting (id, aid, eid, mykey, ...)
     VALUES (1, 'a', 'b', 'a', ...);
INSERT INTO fruit_setting (id, aid, eid, mykey, ...)
     VALUES (2, 'a', 'a', 'a', ...);

START TRANSACTION; /* TX1 */
UPDATE fruit_setting SET ... WHERE
     aid='a' and mykey='a';
                  START TRANSACTION; /* TX2 */
                  UPDATE fruit_setting SET ... WHERE
                        eid IN ('a', 'b') and mykey='a';
```

Listing 3.8: StackOverflow #65519414

to perform a full table scan based on existing data in the table while executing the UPDATE query in TX1. During this process, it tries to acquire an exclusive next-key lock on every primary-key record, including the newly inserted row, and thus gets blocked as the new row is inserted by TX2.

[Pattern-3] Simple Cycles with Two Queries

Description: Pattern-3 deadlocks involve two queries from two transactions, and each query requests multiple locks.

Example: Listing 3.8 shows an example based on StackOverflow question #65519414 [3]. The table schema contains 2 different indexes. One consists of columns `eid` and `mykey`, and the other consists of `aid` and `mykey`. The UPDATE query in TX1 updates the records via searching in the order of index `i_aid_mykey`. Since the two existing rows have the same values for `aid` and `mykey`, the two rows will be accessed in an order based on the values of primary key `id`. Specifically, the query will request an exclusive lock first on the row with `id=1` and then on the row with `id=2`. On the other hand, the UPDATE query in TX2 updates the records via searching in the order of index `i_eid_mykey`. With the two existing rows, it will request exclusive locks on the two rows in an opposite order as the query in TX1, resulting in a deadlock.

3.4.4 Cycles with a Lock Held by Multiple Transactions

Deadlocks involving a lock held by multiple transactions cannot be modeled with the simple cycle already described, and Figure 3.2 shows the deadlock cycle that we come up with

```
CREATE TABLE tradeshistory (
   PRIMARY KEY (id), ...
);
CREATE TABLE trades (
   PRIMARY KEY (id), ...
);

START TRANSACTION; /* TX1 */
INSERT INTO tradeshistory (SELECT
     trades.* FROM trades WHERE id=10);
                    START TRANSACTION; /* TX2 */
                    INSERT INTO tradeshistory (SELECT
                         trades.* FROM trades WHERE id=10);
UPDATE trades SET ... WHERE id=10;
                    UPDATE trades SET ... WHERE id=10;
```

Listing 3.9: StackOverflow #5353877

to model deadlocks involving such locks. In the diagram, transactions `TX1` and `TX2` both hold the same lock on record `R`. Then, they both request the exclusive lock, which conflicts with the lock held by the other transaction, and thus the two transactions get blocked by each other, resulting in a deadlock. Depending on the type of the lock held by multiple transactions, we further divide them into three types. Below, we omit the **Description** paragraph if the pattern name is self-explanatory and we do not have more to add.

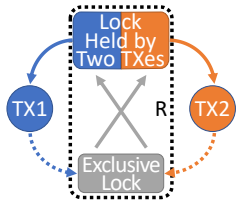

Figure 3.2: A cycle with a lock held by multiple transactions

[Pattern-4] Multiple TXes Holding One Shared Record Lock

Description: The lock held by multiple transactions is a shared record lock, and this is the classical conversion case [140].

Examples: Listing 3.9 shows an example based on StackOverflow question #5353877 [30]. First, the `SELECT` subqueries of `INSERT` in both transactions acquire a shared record lock on the row with `id=10` in table `trades`. Then, the `UPDATE` queries in both transactions ask for an exclusive record lock on the same row, but both get blocked by the shared record lock held by the other transaction. Listing 3.10 shows a similar example from *Lobsters* #39 [24]. The `INSERT` queries in both transactions acquire a shared record lock on the row with `id=1`

```
CREATE TABLE votes ( ...,
   story_id BIGINT(20) NOT NULL, ...,
   FOREIGN KEY (story_id) REFERENCES stories(id);
);
CREATE TABLE stories (
   id BIGINT(20) NOT NULL PRIMARY KEY, ...
)

START TRANSACTION; /* TX1 */
INSERT INTO votes (story_id, ...)
     VALUES (1, ...);
                   START TRANSACTION; /* TX2 */
                   INSERT INTO votes (story_id, ...)
                        VALUES (1, ...);
UPDATE stories SET ... WHERE id=1;
                   UPDATE stories SET ... WHERE id=1;
```

Listing 3.10: *Lobsters* #39

in table `stories`, but this is due to foreign key, which is the same as the case in Listing 3.6.

[Pattern-5] Multiple TXes Holding One Shared Gap Lock

Example: The example from MySQL's official manual in Listing 3.1 as mentioned in Section 2.1 is a Pattern-5 deadlock. In `TX1`, the `INSERT` query acquires an exclusive record lock on the row inserted. In `TX2` and `TX3`, the `INSERT` query asks for a shared record lock during duplicate-key checking because the query inserts the primary key. When `TX1` is rolled back, the `INSERT` queries in `TX2` and `TX3` both get the shared gap lock because the row inserted by `TX1` does not exist anymore. Then, the `INSERT` queries in both transactions ask for the same exclusive insert-intention lock, but both get blocked by the shared gap lock held by the other transaction.

[Pattern-6] Multiple TXes Holding One Exclusive Gap Lock

Description: The lock held by multiple transactions is an exclusive gap lock. Although in the exclusive mode, an exclusive gap lock can be held by multiple transactions simultaneously.

Example: Listing 3.11 shows *MediaWiki* #214035 [8]. With existing data in table page_restrictions, the `DELETE` queries in both transactions acquire an exclusive gap lock on the same range, as the `WHERE` conditions in both queries match no existing rows but fall into the same range. Then, the `INSERT` queries in both transactions ask for an exclusive insert-intention lock on the same range, and they get blocked by the exclusive gap lock held by the other transaction.

Besides `DELETE`, `SELECT FOR UPDATE` or `UPDATE` can also have `WHERE` conditions matching no rows, thus acquiring exclusive gap locks and causing the same type of deadlocks.

```
CREATE TABLE page_restrictions (
   pr_id INT unsigned NOT NULL PRIMARY KEY AUTO_INCREMENT,
   pr_page INT NOT NULL,
   pr_type VARBINARY(60) NOT NULL, ...
)
CREATE UNIQUE INDEX pr_pagetype ON page_restrictions (pr_page,pr_type);

START TRANSACTION; /* TX1 */
DELETE FROM page_restrictions WHERE
     pr_page=125 and pr_type='move';
                    START TRANSACTION; /* TX2 */
                    DELETE FROM page_restrictions WHERE
                         pr_page=150 and pr_type='move';
INSERT INTO page_restrictions
     (pr_page,pr_type,...) VALUES
     (125,'move',...);
                    INSERT INTO page_restrictions
                         (pr_page,pr_type,...) VALUES
                         (150,'move',...);
```

Listing 3.11: *MediaWiki* #214035

3.4.5 Cycles with Lock Queues

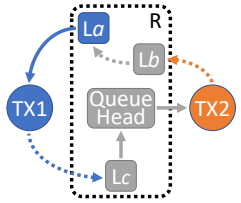

Figure 3.3: A cycle with a lock queue

Each MySQL/InnoDB record internally maintains a queue, and queries requesting locks on the same record are queued in the order these requests are made. Therefore, queries enqueued later need to wait for queries enqueued earlier. Figure 3.3 shows the deadlock cycle that we come up with to model deadlocks involving such wait relationships on lock queues. In the diagram, (1) `TX1` acquires `La`, (2) `TX2` requests `Lb` but gets blocked by `TX1`, and `TX2` is put into the queue corresponding to record `R`, and (3) `TX1` requests `Lc` that conflicts with `Lb` being requested by `TX2`, and thus `TX1` is blocked by `TX2` and put into the same queue after `TX1`. Deadlocks involving lock queues all have three queries, and we further divide such deadlocks based on the query types and lock types involved in the deadlock. We group the examples for Patterns 7, 8, and 9 together as they share the same query pattern. 'X' and 'S' in the following pattern names are lock modes.

```
CREATE TABLE cache_config(
   cid VARCHAR(255) NOT NULL, ...
   PRIMARY KEY (cid)
);

START TRANSACTION; /* TX1 */
DELETE FROM cache_config WHERE cid=1;
                   START TRANSACTION; /* TX2 */
                   DELETE FROM cache_config WHERE cid=1;
INSERT INTO cache_config (cid, ...)
     VALUES(1, ...);
```

Listing 3.12: *Drupal* #2336627

```
CREATE TABLE user_properties (
   up_user INT NOT NULL,
   up_property VARBINARY(255) NOT NULL, ...
)
CREATE UNIQUE INDEX user_properties_user_property ON user_properties (up_user,up_property);

START TRANSACTION; /* TX1 */
DELETE FROM user_properties WHERE
     up_user=1 AND up_property='aaa';
                   START TRANSACTION; /* TX2 */
                   DELETE FROM user_properties WHERE
                        up_user=1 AND up_property='aaa';
INSERT INTO user_properties (up_user,
     up_property, ...) VALUES(1, 'aaa', ...);
```

Listing 3.13: *MediaWiki* #38116

[Pattern-7] DELETE-DELETE-INSERT *Acquiring X Record Lock, X Record Lock, and S Next-key Lock*

[Pattern-8] DELETE-DELETE-INSERT *Acquiring X Record Lock, X Next-Key Lock, and S Next-Key Lock*

[Pattern-9] DELETE-DELETE-INSERT *Acquiring X next-key Lock, X Next-Key Lock, and X Insert-Intention Lock*

Examples: Listing 3.12 shows a Pattern-7 deadlock in *Drupal* #2336627 [10]. In TX1, the DELETE query first acquires an exclusive record lock on the row of cid=1 because it uses the primary key to search for records. In TX2, the DELETE query asks for the same exclusive record lock on the same row but gets blocked. Thus, TX2 is put into a wait queue corresponding to the row of cid=1. Finally, the INSERT query in TX1 wants to insert a record with cid=1. Because cid is the primary key of the table, it asks for a shared next-key lock to check if the primary key value to be inserted exists. This lock cannot be granted because it conflicts with the lock requested by the DELETE query in TX2. Thus, TX1 has to wait for TX2 that is currently the head of lock queue for the row of cid=1, completing the hold-and-wait cycle.

Listing 3.13 shows a Pattern-8 deadlock in *MediaWiki* #38116 [11]. Among all locks that it acquires, the DELETE query in TX1 acquires an exclusive record lock on the unique index satisfying the WHERE condition, as it uses the unique index to search for records. Then, the

```
CREATE TABLE wb_terms (
   term_row_id INT unsigned NOT NULL PRIMARY KEY AUTO_INCREMENT,
   term_entity_id INT unsigned NOT NULL,
   term_entity_type VARBINARY(32) NOT NULL, ...
);
CREATE INDEX wb_terms_entity_id ON wb_terms (term_entity_id);
CREATE INDEX wb_terms_entity_type ON wb_terms (term_entity_type);

START TRANSACTION; /* TX1 */
DELETE FROM wb_terms WHERE term_entity_id=1
     AND term_entity_type='A';
                  START TRANSACTION; /* TX2 */
                  DELETE FROM wb_terms WHERE term_entity_id=1 AND term_entity_type='A';
INSERT INTO wb_terms (term_entity_id,
     term_entity_type, ...) VALUES (1, 'A'...);
```

Listing 3.14: *MediaWiki* #44547

DELETE query in TX2 requests an exclusive next-key lock on the unique index, but it gets blocked due to the aforementioned exclusive record lock held by TX1. Based on comments from MySQL source code, since in a unique secondary index, there may be different delete-marked versions of a record where only the primary key values differ, next-key locks are used on a secondary index when locking delete-marked records. Finally, the INSERT query asks for a shared next-key lock to check if the new row with `up_user=1 AND up_property='aaa'` to be inserted may result in duplicates on the unique index. This lock cannot be granted because it conflicts with the lock requested by TX2. Thus, TX1 again has to wait for TX2.

Listing 3.14 shows a Pattern-9 deadlock in *MediaWiki* #30598 [9]. In this case, the two DELETE queries in both transactions use non-unique indexes to search for records. In TX1, the DELETE query acquires exclusive next-key locks on the two indexes satisfying the WHERE condition because the indexes are non-unique. Then, the DELETE query in TX2 requests the same locks and gets blocked by TX1, and it is put into wait queues corresponding to these two indexes. Finally, the INSERT query in TX1 wants to insert a record sharing the same values with the DELETE query on the non-unique indexes. Since the indexes are not unique, the INSERT query does not need to perform the duplicate key checking, but it will directly request an exclusive insert-intention lock. This lock cannot be granted because it conflicts with the lock requested by TX2. Thus, TX1 has to wait for TX2.

[Pattern-10] `INSERT-SELECT FOR UPDATE-SELECT FOR UPDATE` *Acquiring S Record Lock, X Record Lock, and X Record Lock*

Example: Listing 3.15 shows an example based on StackOverflow question #41015813 [5]. The INSERT query in TX1 inserts a row of `parent_id=1` into table `child`, and it acquires a shared lock on the record satisfying `id=1` in table `parent` because of the foreign-key constraint between these two tables. Then, TX2's SELECT FOR UPDATE query will ask for an

```
CREATE TABLE parent (id INT(11) PRIMARY KEY);
CREATE TABLE child (
   id INT(11) PRIMARY KEY,
   parent_id INT(11),
   FOREIGN KEY (parent_id) REFERENCES parent(id)
);

START TRANSACTION; /* TX1 */
INSERT INTO child (id, parent_id)
     VALUES (10, 1);
                    START TRANSACTION; /* TX2 */
                    SELECT id FROM parent WHERE id=1
                         FOR UPDATE;
SELECT id FROM parent WHERE id=1
     FOR UPDATE;
```

Listing 3.15: StackOverflow #41015813

```
CREATE TABLE cache_config(
   idset_key CHAR(40) NOT NULL,
   member_id INT(11) NOT NULL,
   PRIMARY KEY (idset_key, member_id)
);

START TRANSACTION; /* TX1 */
INSERT INTO reporting_idset
     (idset_key, member_id) VALUES (5, 5);
                    START TRANSACTION; /* TX2 */
                    INSERT INTO reporting_idset
                         (idset_key, member_id) VALUES (5, 5);
DELETE FROM reporting_idset
     WHERE idset_key=5;
```

Listing 3.16: *OpenMRS* #674

exclusive lock on the record satisfying `id=1` in table `parent`. This lock request from `TX2` is blocked by `TX1`. After that, `TX1`'s `SELECT FOR UPDATE` query also asks for an exclusive lock on the same record. This lock request from `TX1` cannot be granted because it conflicts with the lock requested by `TX2`, completing the deadlock cycle.

[Pattern-11] `INSERT`-`INSERT`-`DELETE` *Acquiring X Record Lock, S Record Lock, and X Next-Key Lock*

Example: Listing 3.16 shows *OpenMRS* #674 [13]. In `TX1`, the `INSERT` query inserts a new row in the table `cache_config` and acquires an exclusive record lock on that row. Then, the `INSERT` query in `TX2` tries to insert the same record and asks for a shared record lock on that row for duplicate-key checking. It gets blocked by `TX1` and is put into a wait queue. After that, the `DELETE` query in `TX1` tries to delete records satisfying `idset_key=5`, including the newly inserted record by the previous `INSERT` query in `TX1`. Since `idset_key` is part of the multi-column primary key, it will ask for an exclusive next-key lock on every record satisfying the where condition. This lock cannot be granted as it conflicts with the lock requested by `TX2`, completing the deadlock cycle.

```
CREATE TABLE wbc_entity_usage (
   eu_row_id       BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
   eu_entity_id    VARBINARY(255) NOT NULL,
   eu_aspect       VARBINARY(37) NOT NULL,
   eu_page_id      INT NOT NULL
);
CREATE UNIQUE INDEX eu_entity_id ON wbc_entity_usage ( eu_entity_id, eu_aspect, eu_page_id ); ...

START TRANSACTION; /* TX1 */
INSERT INTO wbc_entity_usage
     (eu_page_id=10, eu_aspect='10', eu_entity_id='10');
                    START TRANSACTION; /* TX2 */
                    INSERT INTO wbc_entity_usage
                         (eu_page_id=10, eu_aspect='10', eu_entity_id='10');
INSERT INTO wbc_entity_usage
     (eu_page_id=9, eu_aspect='9', eu_entity_id='9');
```

Listing 3.17: *MediaWiki* #192349

[Pattern-12] `INSERT`-`INSERT`-`INSERT` *Acquiring X Record Lock, S Next-Key Lock, and X Insert-Intention Lock*

Example: Listing 3.17 shows *MediaWiki* #192349 [14]. The first `INSERT` query in `TX1` acquires an exclusive record lock on both the row being inserted and the unique index with `eu_page_id=10, eu_aspect='10', eu_entity_id='10'`. The `INSERT` query in `TX2` requests a shared next-key lock on the unique index during duplicate-key checking. `TX2` gets blocked by `TX1` and is put into a wait queue. The second `INSERT` query in `TX1` passes the duplicate-key checking, as `eu_page_id=9, eu_aspect='9', eu_entity_id='9'` is not in the table, and it proceeds to request an exclusive insert-intention lock. When existing data in the table makes the insert-intention lock be on the record of `eu_page_id=10, eu_aspect='10', eu_entity_id='10'`, the insert-intention lock requested by `TX1` conflicts with the lock requested by `TX2`. Thus, `TX1` is also blocked by `TX2`.

3.4.6 Discussion

Among those PostgreSQL-lock deadlocks, the 4 from applications are due to cycles with nested transactions, and the 5 from StackOverflow questions are of Patterns 1, 2, and 4. Deadlocks of these patterns can be understood with general SQL knowledge, while deadlocks on MySQL/InnoDB locks are more challenging for application developers to understand.

To help application developers in tackling this challenge, we categorize deadlocks on MySQL/InnoDB locks in fine granularity and provide a concrete example for each pattern that we observe in our deadlock set. We believe the knowledge gained through our examples will be valuable for application developers to understand and diagnose deadlocks that they may encounter, even for those beyond the patterns that we observe. For tool researchers

Table 3.4: Fixing strategies

Fixing Strategies		Total
Fix	Enforcing lock order by changing query order	3
	Omitting unnecessary queries	3
	Omitting unnecessary SELECT FOR UPDATE locks	1
	Removing unnecessary transactions	4
	Avoiding concurrent execution with app-level lock	3
	Avoiding concurrent execution with ordered execution	3
	Avoiding conflicting by changing queries or logic	7
	Avoiding nested transactions	3
	Avoiding using database	1
Reduce	Splitting a large transaction into smaller ones	2
	Reduce the number of resources requested	3
Catch and retry		3

and developers, our results suggest that existing tool support is not sufficient and call for more effort in this area. Specifically, our results on database-lock deadlocks reveal cycle patterns that existing techniques on deadlocks have not accounted for.

3.5 RQ3: Fixes for Database-Lock Deadlocks

Unlike hold-and-wait cycle patterns, the fixing strategies for database-lock deadlocks are much straightforward to understand. Table 3.4 shows the different fixing strategies used for the 36 deadlocks from real-world applications and their corresponding numbers. On the high level, fixing strategies for database-lock deadlocks can be categorized as (1) completely eliminating the possibility of deadlocks, (2) reducing the chance of deadlocks, or (3) adding catch-and-retry.

The majority, i.e., 28 out of 36, of the studied database-lock deadlocks are completely fixed with various strategies. The first three strategies can be viewed as different ways to break the hold-and-wait cycle. The next three strategies can be viewed as different ways to avoid concurrent transactions. The last three strategies are more application-specific. In particular, avoiding nested transactions is only used to fix *Odoo* intra-request deadlocks, and the "avoiding using database" strategy is used when the data can be moved to cache.

5 deadlocks are not completely fixed, but developers either reduce transaction length or reduce the number of resources requested in transactions to reduce the chance of deadlocks. This could happen if a complete fix is too complex, and the chance of deadlocks can be reduced to an acceptable level.

In the remaining 3 cases, developers take the catch-and-retry approach by adding code

to retry transactions on deadlocks, and the chance of deadlocks is likely considered as acceptable.

In the case of StackOverflow questions, 10 of them have accepted answers with fixing strategies proposed. The proposed strategies are no different from what we see in real-world web applications. Since the actual patch being applied in practice is only mentioned in one StackOverflow question, we do not include it in Table 3.4.

3.6 Threats to Validity

Our study may be subject to several validity threats. Next, we describe potential threats and our ways to address them. (1) We may not include all representative web applications. To minimize this threat, we choose popular open-source applications with a significant user base from state-of-the-art studies on web applications, and we search deadlocks in all applications from these studies that are still available. We further include StackOverflow questions to further enrich our understanding of deadlocks on database locks. Our results show the characteristics are similar for database-lock deadlocks from web-application bugs and those based on StackOverflow questions. So the characteristic study results can likely be generalized to other web applications. (2) We may miss relevant bug reports while searching for deadlocks. We mitigate this threat by using keyword search in both bug descriptions and comments together with bug categories and tags. (3) We inspect bug reports manually. To alleviate this threat, each report is examined by at least two authors, and the group discusses the bug report together to reach a consensus.

3.7 Conclusion

In this chapter, we characterize deadlocks from real-world web applications based on the number of HTTP requests and the types of resources involved. For deadlocks on database locks, we further categorize their hold-and-wait cycle patterns and fix strategies. The patterns and concrete examples presented in this chapter can help application developers understand and diagnose deadlocks that they may encounter. Our study results can also guide future research in combating deadlocks in web applications.

CHAPTER

4

A CHARACTERISTIC STUDY ON EXCEPTION HANDLING BUGS IN DATABASE-BACKED WEB APPLICATIONS

4.1 Introduction

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. In modern web applications, exceptions can vary from application-logic-related erroneous conditions, e.g., operating on an invalid bank account number or referencing a null pointer, or environmentally triggered erroneous conditions, e.g., failing to open a file or communicate via network.

Modern Object-Oriented (OO) programming languages, which are usually used to implement web applications, all provide a built-in mechanism for exception handling, which usually contains two parts. Firstly, developers will wrap lines of code, where potential exceptions may occur, in a guarded block so that the exception will be caught when it is raised. Then, there is a handler block after that, where developers should include exception handling logic to properly deal with the exceptional condition. In that case, program execution can continue normally with the code after the exception handling block. The following is

Listing 4.1: Redmine #984

```
def preview
  - issue = @project.issues.find_by_id(params[:id])
  + issue = @project.issues.find_by_id(params[:id]) unless params[:id].blank?
  @attachements = issue.attachments if issue
  @text = params[:notes] || (params[:issue] ? params[:issue][:description] : nil)
  render :partial => 'common/preview'
```

Listing 4.2: Diaspora #26341

```
def post(post, url="")
  begin
   ......
  rescue ActiveRecord::RecordNotUnique => e
  - Rails.logger.info("failed to save received object")
  + Rails.logger.info("failed to save received object: #{e.message}")
  end
end
```

what the exception handling mechanism looks like. In Ruby on Rails, developers place the keyword "rescue" after any code that would probably throw an exception as below:

```
begin
    # This is the guarded block, containing
    # business code that may raise exceptions
    ...
rescue Exception
    # This is the handler block, containing
    # exception handling code
    ...
end
```

The code between "begin" and "rescue" is where possible exceptions might occur. If an exception occurs, and it matches or is a subset of the exception types listed after "rescue", the code block between "rescue" and "end" will be executed to take corresponding actions for exceptional scenarios.

With the exception handling mechanism described above, correctly handling exceptions requires two conditions: (1) developers need to put business code into a guarded block to catch the expected exceptions, and (2) the exception handling logic in the handler block can gracefully handle the caught exception to ensure the web application remains robust and user friendly. If an exception occurs but there is no guarded block to catch it, the request handler where the exception is raised will crash. This means it will terminate the execution of the remaining code abnormally and directly return the stack traces and error messages

to users on the web page.

For instance, as shown in code 4.1, the function `preview` reads data from the database by calling the function find_by_id with the input id value. This function automatically constructs a SELECT SQL query and sends it to the database. It works fine most of the time but crashes when the input value is empty as it raises an `ActiveRecord::StatementInvalid` exception but it is not caught. On the other hand, in code 4.2, although an exception has already been captured and handled, a developer still filed a bug report complaining that the logged exception message was not informative enough and needed more details. Therefore, the developers refined the log statement in the exception handler block to include more information when handling the exception.

In this paper, we refer to bugs caused by either *missing handling of a certain type of exceptions* or *improper exception handling* as *Exception Handling (EH) bugs*.

Although many recent studies have spent research efforts on bugs in database-backed web applications, they only focus on different bug types other than exception handling bugs, e.g., performance bugs and concurrency bugs [135, 136, 137, 150, 154, 163, 167]. Most of these studied bugs do not involve exceptions or exception handling. Only a few bugs are related to exceptions, but they just cover certain exceptions that are especially related to concurrency. For example, the aforementioned `ActiveRecord::StatementInvalid` exception is unlikely to be raised in a performance bug or concurrency bug. The previous work fails to provide a comprehensive view of EH bugs in database-backed web applications.

Some existing studies have been conducted on error handling used in operating system (OS) kernels and file systems [88, 130, 142]. However, OS kernels and file systems are implemented in C language while web applications are usually implemented in Object-Oriented (OO) programming languages, such as PHP, Java, and Ruby. C language is different from these OO programming languages as it does not provide structured error handling primitives, i.e., C language does not have a throw-catch-handle mechanism as the OO programming language does to handle exceptions. C language handles exceptions by just checking the returned exception code, which is usually an integer value to indicate success or failure. In other words, the gained insights of exception handling used in OS cannot be generalized to EH bugs in database-backed web applications.

Yuan et al. [172] studied catastrophic failures reported in distributed systems that are implemented in Java. As an OO programming language, Java uses the language-provided exception handling mechanism to separate the functional code from exception handling logic. Their paper found that some catastrophic failures are triggered by wrong error handling logic implemented in the exception handler block. However, the failures that they studied are from distributed systems, not web applications, and they did not mention if

there are failures that are caused by missing handling of certain exceptions in their paper. It's not clear what the characteristics of missing exception handling and improper exception handling bugs are in database-backed web applications.

In general, the previous work fails to provide a comprehensive understanding of exception handling bugs. To complement the current state of the art and help developers to have a better understanding of how to properly combat exception handling bugs, we conduct a characteristic study on them.

We start our study with the following research question:

- **RQ1:** How many EH bugs are related to missing exception handling of certain exceptions, and how many EH bugs are related to improper exception handling?

To answer this question, we conduct a comprehensive empirical study on a set of 7 real-world Ruby on Rails web applications. We choose Ruby on Rails as it is a popular ORM framework [19] developers use to build modern database-backed web applications nowadays. We do keyword searches in the bug tracking systems of these web applications and collected a total number of 214 exception handling bugs. Among these collected bugs, 199 of them are no exception handling bugs while only 15 of them are improper exception handling bugs. Our results suggest that no exception handling bugs are much more common and developers should be more careful about them.

We then proceed with the following research questions to further categorize exception handling bugs. For no exception handling bugs:

- **RQ2.1:** What are the exception types involved in the bugs?
- **RQ2.2:** What are the root causes and triggering conditions?
- **RQ2.3:** What are the strategies used to fix these bugs?

For improper exception handling bugs, we have the same characteristic research questions:

- **RQ3.1:** What are the exception types involved in the bugs?
- **RQ3.2:** What are the root causes and triggering conditions?
- **RQ3.3:** What are the strategies used to fix these bugs?

To answer these research questions, we study the collected sets of no exception handling and improper exception handling bugs respectively. On **RQ2.1/2/3**, we divide unhandled exceptions into different categories based on whether the exception is raised when the application is accessing an external resource or not. Among these exceptions, 110 of them

are raised when accessing an external resource, i.e., database, file, or remote server. Our result suggests that most of the exceptions involved in our collected exception handling bugs are related to external resources. Therefore, developers should pay more attention to exception handling when implementing functionalities that will access an external resource. No exception handling bugs are due to raised exceptions not being handled, and we then summarize the exceptional conditions that will trigger the raise of these unhandled exceptions. For fixing strategies, we note that only 73 bugs are fixed by using a built-in exception handling mechanism, while most of the collected bugs can be fixed by changing the semantics or modifying the external resources. On **RQ3.1/2/3**, we find that similar to no exception handling bugs, most of the involved exceptions in improper exception handling bugs are related to external resources. Besides, we also characterize the 15 improper exception into 3 root causes and summarize 5 strategies used to fix these bugs.

We expect our results to provide a deep understanding of the characteristics of exception handling bugs from real-world database-backed web applications, which can benefit not only application developers but also tool developers. For tool developers, our results can help to guide the design of tools for different purposes, e.g., developing tools to detect potential unhandled exceptions or to automatically fix exception handling bugs with our results on **RQ2** and **RQ3**.

4.2 Methodology

In this section, we describe our methodology on how we collect and study exception handling bugs.

We choose database-backed web applications written in Ruby on Rails, which is a very popular web application framework [19] and widely used projects hosted on Github [36]. Many well-known web applications are built on top of Ruby on Rails, such as Github, Airbnb, Hulu, Shopify, etc [2]. We select 7 popular web applications, i.e., Redmine [34], Onebody [32], Tracks [38], Lobsters [23], OpenStreetMap [33], Spree [35], and Diaspora [16]. These web applications are all widely used in existing studies on performance bugs in web applications [150, 163, 167], concurrency bugs in web applications [135, 136, 137, 160], and data constraints in web applications [165]. Table 4.1 lists the applications used in our study. They are all popular projects with more than 1000 stars on Github.

We then follow the methodology taken by these aforementioned existing studies together with some other existing studies on concurrency bugs [78, 115, 157], while collecting and studying bug reports related to exception handling.

To collect exception handling bugs, we first search keywords that are related to exception

Table 4.1: Applications and bugs used in our study.

Applications	Stars	Commits	Contributors	Bugs
Redmine	5168	18251	8	108
Onebody	1398	5203	50	5
Tracks	1200	4953	84	9
Lobsters	3954	2845	188	9
OpenStreetMap (OSM)	2055	14234	215	10
Spree	12709	24963	847	37
Diaspora	13369	20962	371	36

handling, e.g., “exception,” “raise,” “throw/thrown,” and “rescue”, in the bug tracking system of each applications. After the keyword search, we get around 1000 results. We then manually filter out bugs that are labeled as not closed, duplicate, or will not fix, and bugs with keywords used in a context that is not related to exception handling. Our final collection contains a total number of 214 bugs, and all these bugs have sufficient information for us to understand. Table 4.1 shows the number of collected bugs in each application.

To answer all these proposed research questions, we analyze each bug’s description, discussions, commit messages, code, and all other available resources to help us determine the characteristics results. At least two persons will first individually conduct the study and answer the characterization questions mentioned in the last section mainly based on manual investigation leveraging all available resources we can find in each bug report. Then, we will cross-check each other’s results to reach an agreement on the characterization results. This process helps to further reduce threats to credibility and validity.

Table 4.2: Numbers of no exception handling and improper exception handling bugs.

Applications	Number of no EH bugs	Number of improper EH bugs
Redmine	103	5
Onebody	4	1
Tracks	9	0
Lobsters	7	2
OpenStreetMap (OSM)	7	3
Spree	36	1
Diaspora	33	3
Total	199	15

4.3 RQ1: No Exception Handling V.S. Improper Exception Handling

In our study, we first divide all 214 exception handling bugs into two groups. Table 4.2 shows the numbers of no exception handling bugs and improper exception handling bugs in each web application.

As we can see from the table, most of our collected bugs (199 out of 214) are caused by no exception handling. The results suggest that developers are very prone to ignoring potential exceptions so that they will miss catching them during the implementation. This is not surprising, as in a modern web application, the business logic can be very complex because the application can contain many web pages and each page can involve various functionalities. For each functionality, there can be various conditions in the code where potential exceptions may be raised due to the complexity. Therefore, it is challenging for developers to take all of the exception conditions into consideration and handle them during the implementation. The overall numbers of no exception handling bugs and improper exception handling bugs suggest that developers should pay much more attention to no exception handling bugs. We will further discuss their characteristics in Sections 4.4 and 4.5.

4.4 RQ2: Characteristics On No Exception Handling Bugs

In this section, we focus on no exception handling bugs and discuss the involved exception types, the root causes, and fixing strategies.

4.4.1 Exception Types

We have collected a total number of 199 no exception handling bugs. After analyzing the descriptions of collected bugs, developers' discussions, commit messages, code comments, and other available resources, we summarize the exception types based on the unhandled exception of each bug. On the high level, we divide these exceptions based on whether an exception is raised when accessing an external resource, e.g., database, and derive 6 resource types. Table 4.3 shows the exception resource types and the exception types for each. Note that, the numbers in the parentheses are the numbers of exceptions involved in improper exception handling bugs. We will discuss them in Section 4.5. Below, we will explain each exception type we have categorized.

Table 4.3: Overall results on exception types. Numbers in the parentheses are for improper exception handling bugs.

	Redmine	Onebody	Tracks	Lobsters	OSM	Spree	Diaspora	Total
Number of bugs studied	103 (5)	4 (1)	9	7 (2)	7 (3)	36 (1)	33 (3)	199 (15)
Database Exceptions								
ActiveRecord::StatementInvalid	31	0	3	2	0	7	1	44
ActiveRecord::RecordInvalid	1 (1)	0 (1)	2	0	0	1	1	5 (2)
ActiveRecord::RecordNotFound	2	1	0	0	2 (1)	2	3 (1)	10 (2)
ActiveRecord::RecordNotUnique	2	1	0	1	0	2	3 (1)	9 (1)
ActiveRecord::RangeError	1	0	0	0	0	0	0	1
ActiveRecord::InvalidForeignKey	2	0	0	0	0	0	1	3
ActiveRecord::NotNullViolation	2	0	0	2	0	0	0	4
Custom Exception	2	0	0	0	0	0	0	2
File Exceptions								
LoadError	0	0	0	0	0 (1)	2	2	4 (1)
Errno::ENOENT	2 (1)	0	0	0	0 (1)	0	0	2 (2)
File-related API exception	5 (2)	0	0	0	0	0	3	8 (2)
Network Connection Exceptions								
SocketError	2	0	0	0 (1)	0	2	3	7 (1)
TimeoutError	0	0	0	0	1	1	0	2
Connection-related API Exceptions	1 (1)	1	0	0 (1)	0	4	2	8 (2)
Custom Exception	1	0	0	0	1	0	0	2

Table 4.3 (continued)

	Redmine	Onebody	Tracks	Lobsters	OSM	Spree	Diaspora	Total
Grammar Exceptions								
NoMethodError	28	1	4	0	1	10 (1)	3	47 (1)
ArgumentError	2	0	0	0	0	3	1	6
TypeError	7	0	0	0	1	0	1	9
NameError	0	0	0	0	0	0	1	1
String Checking Exceptions								
EncodingError	7	0	0	0	0	0	0	7
RegexpError	2	0	0	0	0	1	0	3
URI::InvalidURIError	2	0	0	1	0	0	0	3
Condition Checking Failure Exceptions								
Custom Exception	1	0	0	1	1	1	8 (1)	12 (1)

Table 4.4: Overall results on conditions to trigger exceptions in no exception handling bugs.

	Redmine	Onebody	Tracks	Lobsters	OSM	Spree	Diaspora	Total
Number of bugs studied	103	4	9	7	7	36	33	199
Database Exceptions								
No record exist	2	1	0	0	2	2	3	10
Constraint violation - uniqueness(concurrent)	1	1	0	0	0	2	3	7
Constraint violation - uniqueness(sequential)	1	0	0	1	0	0	0	2
Constraint violation - foreign key	2	0	0	0	0	0	1	3
Constraint violation - null	2	0	0	2	0	0	0	4
Constraint violation - field length	7	0	0	0	0	0	0	7
App side constraint violation	1	0	2	0	0	1	1	5
Inconsistency between code and schema	4	0	0	0	0	4	0	8
Deadlock	0	0	0	0	0	1	0	1
Invalid query - invalid input	9	0	1	0	0	1	1	12
Invalid query - invalid encoding	4	0	0	0	0	0	0	4
Invalid query - unsupported usage	4	0	0	1	0	0	0	5
Invalid query - invalid usage	6	0	2	1	0	1	0	10
File Exceptions								
File doesn't exist	3	0	0	0	0	2	4	9
Cannot parse file	4	0	0	0	0	0	1	5
Network Connection Exceptions								
Connection failure	4	2	0	0	1	7	5	19

Table 4.4 (continued)

	Redmine	Onebody	Tracks	Lobsters	OSM	Spree	Diaspora	Total
Grammar Exceptions								
Nil object	19	1	4	0	1	9	3	37
Non-defined method	9	0	0	0	0	1	0	10
Wrong arguments	2	0	0	0	0	3	1	6
Illegal types for a operation	7	0	0	0	1	0	1	9
Wrong variable name	0	0	0	0	0	0	1	1
String Checking Exceptions								
Invalid string	2	0	0	0	0	1	0	3
Invalid encoding	7	0	0	0	0	0	0	7
Invalid regular expression	2	0	0	1	0	0	0	3
Condition Checking Failure Exceptions								
Condition checking fails	1	0	0	1	1	1	8	12

After going through all the collected bugs, we find that database, file, and remote server are the three types of external resources the code is attempting to access when an exception is thrown. Apart from those exceptions that are related to the three external resources, we also see exceptions that are thrown without accessing external resources and are related to grammar check, string check, and condition check. Therefore, we categorize these collected exceptions into such 6 types:

- **Database exceptions.** Exceptions in this category are primarily related to the use of the ActiveRecord library, which is a crucial component of the Ruby on Rails ORM framework. Developers utilize ActiveRecord APIs, which automatically translate into SQL queries, to manipulate data stored in the database. However, if these queries result in database execution failures, then failure-specific exceptions are thrown.

- **File exceptions.** When an application is attempting to access a file, but if the file does not exist or cannot be parsed due to invalid contents, the file access operation fails. Meanwhile, an exception indicating the failure will be thrown, including language-provided exceptions, e.g., `Errno::ENOENT` and `LoadError`, or API-specific exceptions based on what APIs a web application uses.

- **Network connection exceptions.** Some web applications attempt to establish connections with external servers to make use of services provided by third parties, e.g., fetching content from AWS or Twitter, by calling APIs provided by these service providers. These connections sometimes cannot be successfully established and then network connection failure exceptions will be raised.

- **Grammar exceptions.** This category encompasses exceptions that are related to the programming language's grammar. They are all Ruby on Rails' built-in exceptions and typically are raised at runtime due to programming errors that violate the language's grammar. For example, developers often encounter the `NoMethodError` exception when attempting to call a method on an object that is *nil*.

- **String checking exceptions.** We also have a category of exceptions that are related to string checking. For example, a web application may try to access a URL. Before sending the URL request, the application has code to check if the URL string is valid and will raise an exception if the string is malformed.

- **Condition checking failure exceptions.** All the exceptions in the last category are custom exceptions. These exceptions are intentionally created with a descriptive self-defined name and clear exception message. Once a certain if condition checking fails,

Listing 4.3: Diaspora #6375

```
def try_to_login(login, password, active_only=true)
  ......
  + begin
      handle_http_response(response)
  + rescue HTTPFailure => e
  +   Rails.logger.error(e.message)
  + end
  ......
end

def handle_http_response(response)
  if !response.success
    raise HTTPFailure, "unsuccessful response code: #{response.status}"
end
```

Listing 4.4: Redmine #34071

```
def try_to_login(login, password, active_only=true)
  + begin
      try_to_login!(login, password, active_only)
  + rescue AuthSourceException => e
  +   logger.error "An error occured when authenticating #{login}: #{e.message}"
  +   nil
  + end
end

def try_to_login!(login, password, active_only=true)
  ......
  attrs = AuthSource.authenticate(login, password)
  ......
end

def authenticate(login, password)
begin
  ......
rescue Net::LDAP::Error, Errno::ECONNREFUSED, SocketError
  raise AuthSourceException.new("#{auth_method_name}: #{e.message}")
end
```

then the corresponding custom exception will be raised. As shown in code 4.3, a self-defined `HTTPFailure` exception is raised if the response object's success field's value is false. Note that not all custom exceptions are condition checking failure exceptions. In code 4.4, the custom exception `AuthSourceException` is a re-thrown exception as the result of catching and handling the network connection failure exceptions. We take this exception as a network connection exception instead of a condition checking failure exception because it is raised when failing to connect to a remote server and does not involve an if condition checking.

Discussions. According to table 4.3, approximately 38% of the exceptions (76 out of 214) are related to database issues. This high occurrence of database exceptions is unsurprising given the nature of database-backed web applications. Such applications heavily interact

with databases through SQL queries while users browse web pages, often resulting in bugs that trigger database-related exceptions during query execution. Moreover, developers frequently rely on Object-Relational Mapping (ORM) frameworks to abstract away the complexity of crafting SQL queries manually, inadvertently increasing the likelihood of exceptions. Therefore, developers must pay closer attention to functions responsible for querying databases when developing web applications in the future.

Another notable observation is the prevalence of language-related exceptions, accounting for almost 31% of cases (63 out of 214). Of these, the `NoMethodError` exception stands out, representing over 73% of language-related exceptions. This underscores the importance of careful examination of such exceptions during web application development.

Besides, developers also need to pay attention to custom exceptions. Although there are only 16 custom exception bugs in our collected set, we still think it is easier for developers to ignore them while developing an application because developers may be less familiar with them compared to other common exceptions.

In general, developers often overlook handling database-related and language-related exceptions when implementing functionalities in database-backed web applications. It is important for developers to recognize and address these potential scenarios that may give rise to exceptions in the future.

4.4.2 Root Causes and Triggering Conditions

The root cause for all no exception handling is pretty straightforward, i.e., missing exception handling of certain exceptions. In order to provide guidance for testing to trigger no exception handling bugs and help developers to be aware of the common exceptional conditions in advance and to avoid them during the implementation, we then summarize what are the common conditions to trigger the exceptions by characterizing these bugs. The table 4.4 shows the overall results of exception-raising conditions of bugs that are caused by no exception handling. Among all the 6 exception types, database-related and grammar-related exceptions have the most triggering conditions. We summarize 6 triggering conditions for database-related exceptions and 5 triggering conditions for grammar-related exceptions. We will first introduce the 6 triggering conditions for database-related exceptions:

- **No record exist.** Web applications will call `find()` function to get a record by the given primary key. If the record does not exist in the database, then the `find()` function will raise a `ActiveRecord::RecordNotFound` exception, e.g, code 4.5 is such a case.
- **DB-side Constraint violation.** Violating table schemas' constraints will lead databases to raise exceptions. There will usually be some constraints defined on a table schema,

Listing 4.5: Spree #1860

```
def show
  - @taxon = Taxon.find(params[:id])
  + begin
  +   @taxon = Taxon.find(params[:id])
  + rescue ActiveRecord::RecordNotFound
  +   render_404
  + end
  params[:taxon] = @taxon.id
  @searcher = Spree::Config.searcher_class.new(params)
  @products = @searcher.retrieve_products
end
```

e.g., uniqueness constraint, foreign key constraint, not-null constraint, and column length constraint. When the executing query violates these constraints, the database will fail to execute such a query.

- **App-side constraint violation.** Ruby on Rails provides model validations to ensure that only valid data are saved to the database. When calling a function, like `create!()`, `save!()`, or `update!()` to send an INSERT or UPDATE query statement to the database, validations will be triggered to run before these queries are sent to the database. If any validations fail, the INSERT or UPDATE operation will not be performed and a `ActiveRecord::RecordInvalid` exception will be raised. As shown in the code 4.6, the Project model defines a validation named *validates_uniqueness_of*, which is used to ensure the project's name to be saved is unique in the database. Originally, when calling save! function in the "convert_to_project" to INSERT a project record to the database, the validation fails as the project's name already exists in the database, leading to an exception. Note that, this is different from the DB-side constraint violation, which raises an exception after a query is executed and violates the constraint on the database side. Bugs in this category raise exceptions before a query is executed.

- **Invalid query.** This is the situation where the database fails to execute an invalid query statement. More specifically, we summarize four reasons that can lead to an invalid query statement: 1. Empty input leads a dynamically constructed query to have a grammar violation. The example mentioned in the code 4.1 is such a case. 2. The input value's encoding type is not supported by the database engine, causing query execution failures. 3. Using SQL grammar or SQL functions that are not supported by a database engine. For example, using `CONCAT()` function to concatenate string is not allowed in SQLite. 4. Developers use wrong grammar, e.g., an exception is raised because the expression used after ORDER BY does not appear in the select list when

Listing 4.6: Tracks #1265

```
def convert_to_project
  ......
  project = Project.new(:name => @todo.name, ......)
  + unless project.invalid?
      project.save!
  + else
  +   flash[:error] = "Project name already exists"
  +   redirect_to request.env["HTTP_REFERER"] || root_url
  + end
  ......
end

class Project < ActiveRecord::Base
  ......
  validates_uniqueness_of :name
  ......
```

the SELECT DISTINCT keywords are specified in a query statement.

- **Inconsistency between code and schema.** Developers define information like table names or column names in a schema and may update such information from time to time during the development. However, sometimes developers may forget to update the use of these schema information in the code accordingly, which will lead to exceptions. For example, when an application sends a query to database to retrieve data with an old column name, which has already been changed in the schema. Therefore, an exception will be raised as the database cannot find the matched column.

- **Deadlock.** When concurrent transactions are waiting for each other to give up locks on database lows and thus none of them can proceed, leading to a deadlock situation. When a deadlock is detected by the database engine, it resolves the deadlock by choosing one transaction as the victim, rolling back the transaction, and raising an exception.

For grammar-related exceptions, we have the 5 trigger conditions as follows:

- **Nil object.** Some exceptions are raised when people call a function in the form of "bar.foo" in the code, where "bar" is the name of an object instance and "foo" is the name of a function. This usually will not be a problem. However, sometimes developers forget to check if the object is a nil object before using it. Thus, when calling a method on a nil object leads to a `NoMethodError` exception.

- **Non-defined method.** Besides calling a method on a nil object, another root cause for raising the NoMethodError is when an object calls a method that does not belong to the object's class.

- **Wrong arguments.** Exceptions are raised when the types or number of arguments provided to a method is wrong.
- **Illegal types for an operation.** Exceptions are raised when trying to perform operations that are not allowed between the given types. For example, developers try to do string concatenation between two objects with the operator "+", but the first object is String while the second object is nil. Then a `TypeError` exception will be raised.
- **Wrong variable names.** When you use a variable, but its name is invalid or undefined, then an exception will be raised. These variables usually have been defined earlier, but developers make typos while using them, leading to undefined variable names.

As for other exceptions, the triggering conditions are less complex compared to database and grammar exceptions. Below, we list the triggering conditions for file, network connection, string, and condition check exception, respectively.

File operation failure. Developers use functions to do read or write operations on a file. However, if the file does not exist or the file content is malformed, the functions will fail to operate on the file and raise a file operation failure exception.

Network connection failure. Applications try to establish a connection to a server. However, if the connection cannot be successfully established as the server does not respond and timeouts, the function will raise a connection failure exception.

String validation failure. String object is widely used in web applications, and thus these string-related exceptions will be raised if the validations that are used to ensure a string correctly formed fails. We find 3 types of validation failure among the string-related exceptions: (1) check if an URI string is correct, (2) check if the string's value conforms to the application's supported encoding type, or (3) check if a defined regular expression is valid.

Condition checking failure. Developers intentionally create checking conditions in the application code. During the runtime, once this certain checking condition is met, a custom exception with a very descriptive name and a very clear exception message will be raised.

Discussions. Most constraint violations will lead the application to raise a `ActiveRecord::StatementInvalid` exception, except for the Uniqueness Constraint, which will cause the `RecordNotUnique` exception. It's an exception raised when duplicate data is being inserted into database. As for duplicate data insertion, we have 7 cases that are caused by 2 concurrent requests that are trying to insert the same data. We also have another 2 uniqueness constraint violation cases where exceptions are raised when one request is inserting the data but the same data has already existed.

Yang et al. [165] studied both database-side and app-side constraints. While they mentioned that database-side constraint violation will lead to the raise of an `ActiveRecord::StatementInvalid` exception, they were not aware of the failure of app-side constraint check, i.e., Ruby on Rails' validation, will also lead to the application's throwing an `ActiveRecord::RecordInvalid` exception. Balis et al. [160] also studied app-side constraints used in web applications. However, they only focused on whether Ruby on Rails' validation can provide concurrency control.

Besides, constraint violations, wrong query grammar, and inconsistency between schema and code will also cause the `ActiveRecord::StatementInvalid` exception to be thrown. In essence, all these conditions make the query sent to the database a failed execution. Therefore, developers should get themselves familiar with database knowledge, especially with ORM frameworks, which will automatically translate function calls into SQL queries.

During the implementation, developers should think about "empty" cases and how to handle them in order to avoid exception handling bugs. By "empty" cases, we mean that the code will try to read some empty resources. For example, using `find()` function to get a nonexistent record, constructing a query with empty input, loading a file that is not created yet, and calling a function on an empty object. We have a total number of 64 such bugs, which is not insignificant. Developers should be aware of handling these scenarios during implementation or come up with such corner cases during the testing.

4.4.3 Fixing Strategies

To categorize the fixing strategies used for no exception handling bugs, we manually checked the final committed patch of each bug. The table 4.5 shows the overall results for no exception handling bugs. As we mentioned earlier in Section 4.1, modern web applications are built with programming languages that provide built-in exception handling mechanisms. Therefore, it's intuitive that developers can handle the exceptions by using the mechanism to catch and handle the exceptions. However, in these collected real-world bugs, we also find fixes without using an exception handling mechanism and can be categorized as (1) semantics changes, (2) modify queries, or (3) modify external resources, such as alter table schema or modify file contents. We will start with explaining the strategy using built-in exception handling first, i.e., catch and handle.

Catch and handle. In Ruby on Rails, developers can use the language-provided begin-rescue-end block to catch the exception and handle it properly, so that the program can continue and exit successfully.

We have a total number of 73 bugs that were fixed by using exception handling mech-

Table 4.5: Overall results on how no exception handling bugs are fixed.

	Redmine	Onebody	Tracks	Lobsters	OSM	Spree	Diaspora	Total
Number of bugs studied	103	4	9	7	7	36	33	199
Database Exceptions								
Catch and handle - Assign a vaue	4	0	1	0	0	0	1	6
Catch and handle - Logging	2	2	0	0	0	0	2	6
Catch and handle - Do nothing	0	0	0	0	0	0	1	1
Catch and handle - Render error message	2	0	0	0	0	2	3	7
Semantics - Add condition check	10	0	2	1	0	1	1	15
Semantics - Replace method	0	0	0	0	0	0	1	1
Semantics - Change encoding type	4	0	0	0	0	0	0	4
Semantics - Other	0	0	0	1	0	2	0	3
Modify query	15	0	2	2	0	3	0	22
Alter schema	6	0	0	1	0	3	0	10
File Exceptions								
Catch and handle - Assign a vaue	1	0	0	0	0	0	0	1
Catch and handle - Logging	3	0	0	0	0	2	3	8
Catch and handle - Do nothing	0	0	0	0	0	0	1	1
Semantics - Replace method	1	0	0	0	0	0	0	1
Semantics - Other	1	0	0	0	0	0	0	1
Modify file content	1	0	0	0	0	0	1	2
Network Connection Exceptions								
Catch and handle - Assign a vaue	2	1	0	0	1	6	2	12
Catch and handle - Logging	2	0	0	0	0	1	3	6
Catch and handle - Render error message	0	0	0	0	1	0	0	1

Table 4.5 (continued)

	Redmine	Onebody	Tracks	Lobsters	OSM	Spree	Diaspora	Total
Grammar Exceptions								
Catch and handle - Assign a vaue	1	0	1	0	0	0	0	2
Catch and handle - Logging	0	0	0	0	0	1	3	4
Semantics - Add condition check	20	1	1	0	1	9	2	34
Semantics - Create method	1	0	2	0	0	1	0	4
Semantics - Replace method	8	0	0	0	1	0	0	9
Semantics - Change variable type	5	0	0	0	1	1	0	7
Semantics - Fix arguments	2	0	0	0	0	1	0	3
Semantics - Other	0	0	0	0	0	0	1	1
String Checking Exceptions								
Catch and handle - Assign a vaue	1	0	0	0	0	0	0	1
Catch and handle - Logging	1	0	0	0	0	0	0	1
Catch and handle - Do nothing	1	0	0	0	0	1	0	2
Semantics - Change encoding type	6	0	0	0	0	0	0	6
Semantics - Change regular expression	2	0	0	1	0	0	0	3
Condition Checking Failure Exceptions								
Catch and handle - Assign a vaue	0	0	0	0	0	0	1	1
Catch and handle - Logging	1	0	0	1	0	0	6	8
Catch and handle - Do nothing	0	0	0	0	1	0	0	1
Catch and handle - Render error message	0	0	0	0	0	1	1	2

Listing 4.7: Redmine #2126

```
def redirect_back_or_default(default)
  back_url = CGI.unescape(params[:back_url].to_s)
  if !back_url.blank?
    - uri = URI.parse(back_url)
    - if (uri.relative? || (uri.host == request.host)) &&
         !uri.path.match(%r{/(login|account/register)})
    -   redirect_to(back_url) and return
    + begin
    +   uri = URI.parse(back_url)
    +   if (uri.relative? || (uri.host == request.host)) &&
          !uri.path.match(%r{/(login|account/register)})
    +     redirect_to(back_url) and return
    +   end
    + rescue URI::InvalidURIError
    end
    redirect_to_default
  end
```

anism. Based on actions that are performed in the handler block, the catch-and-handle mechanism can be further categorized into 4 types, i.e., do nothing, logging, assign a value, and render error:

- **Do nothing.** There is nothing in the handler block when an exception is caught. Developers do not need to do anything to enable the function to continue to run the consequent code after the handler block. In code 4.7, the code just catches the exception to ensure the function can exit properly and does nothing else. The main reason for developers to do that is because the exception scenario is not critical or doing nothing will not lead the subsequent execution to an erroneous status. For example, in the code snippet, there is already a redirect_to_default at the end of the function, which can be used to return value for any unsuccessful URI parsing.

- **Logging.** The handler will have code to log the error information, such as exception name, exception message, or function information. The logged data can provide details about the exception, which is crucial for helping developers diagnose and troubleshoot the exceptions.

- **Assign a value.** In this case, the developers need to perform some specific operations in the handler so that the function can correctly continue to execute the code after the handler. For example, the function will return a value in a normal workflow when the function finishes properly, but it encounters an exception during the execution. To correctly handle the exception, developers need to assign a value for return so that the program after the handler block can continue to be executed normally.

- **Render error.** In web applications, sometimes developers will render errors to the user end. Unlike logging, the error message needs to be as detailed as possible to help developers to understand the exception, error that is to be rendered to users needs to be processed and easy for users to read. With rendered messages, users can easily find what is wrong with their operations and how to proceed.

The fixing strategies of the remaining 126 bugs do not involve an exception handling mechanism. Below, we detail each fixing strategy used to resolve the exception.

Semantics changes. Most of the remaining 126 can be fixed by making proper semantics changes. There are various fix strategies that can change the application logic, and we are going to discuss some major approaches as follows:

- **Add check conditions.** This fix is to add checking conditions to prevent illegal cases that will lead to exceptions if it is used in the subsequent execution, e.g., calling a method on a null object, or using an input whose value is out of proper range. Therefore, before the code proceeds, developers can have the if-conditions to check if exceptional conditions are met first and deal with these exceptional cases properly, e.g., not execute the subsequent code or initialize the object with a valid value then proceed, so that the function can run the following code without any interruptions.

- **Change method.** Besides calling a method on an empty object, the `NoMethodError` exception can also be raised when calling a method that does not belong to the object's class, i.e., there is no definition of such a function in the object's class. To fix such exceptions, the intuitive way is just to replace the method with a method that has the same functionality but has been defined in the object's class.

- **Create method.** One can also directly create the method in that class if the method being called does not exist.

- **Change encoding type.** This strategy is used to fix those encoding-related exceptions. To fix these exceptions, developers need to unify the encoding, e.g., they can change the encoding type to UTF-8, which is the most commonly used encoding format.

- **Change regular expression.** For regular expression exceptions, the fix is to modify the regular expression so that it can match the string pattern that developers expect.

- **Fix arguments.** Developers just pass the correct arguments to the method to fix the bug.

- **Change variable types.** On mismatched-type exceptions, a developer will cast the object to a matched type to make the code proceed without raising an exception.

Other than the aforementioned catch-and-handle and semantics changes, we also have another 3 fixing strategies:

Modify query. As we mentioned in section 4.4.2, many database-related exceptions are raised because the database engine fails to execute an invalid query, e.g., the example used in fig 4.1. To fix such exceptions, developers need to modify the query in code and its related ORM APIs so that a valid can be executed by the database.

Alter table schema. We also find some database-related exceptions are raised due to a query's execution violating the table schema's constraint, e.g., the string to be saved is out of the defined field's length, or the inconsistency between the code and the table schema in section 4.4.2. To fix these exceptions, developers can modify the schema definition to pass the query execution.

Modify file content. Developers can correct the file content so that the file can be read and parsed successfully.

Discussions. To our surprise, the majority of the collected no exception handling bugs actually can be fixed without an exception handling mechanism. It is not always the case that exceptions need to be explicitly caught and handled by declaring the begin-rescue-end block. Only around 37% of our collected bugs are fixed by using the exception handling mechanism provided by the language. More specifically, bugs that are related to file, network, and self-defined exceptions are more likely to be fixed by capturing and handling the raised exceptions, while bugs in other exception types are more likely to be fixed by strategies other than capturing and handling.

For database-related exceptions, most raised exceptions are caused by failed query executions that are related to syntax error, wrong grammar, or constraint violation, and thus such exceptions can be easily fixed by modifying the query or altering the table schema definitions. Only for exceptions that are caused by retrieving a record that does not exist or executing a query that violates uniqueness constraint, i.e., `ActiveRecord::RecrodNotFound` exception and `ActiveRecord::RecordNotUnique` exceptions, developers tend to fix them using exception handling.

Almost all grammar-related (58 out of 63) no exception handling bugs do not need to be fixed with a "rescue" block, i.e., they can be fixed by changing the code directly. Even for the bugs that are fixed using an error handling mechanism, we think they actually can be fixed by changing the semantics instead of using exception handling. Most fix strategies used to fix grammar-related exceptions are to add a checking condition to preclude the exception case.

All network-connection-related and most file-related (10 out of 14) no exception handling bugs are fixed by using exception handling. These exceptions are environmentally triggered erroneous conditions, i.e., they are caused mainly because of the erroneous external resources. For example, the application fails to read an invalid file or establish a connection with an unresponsive server. Moreover, they mainly call some APIs that will raise these exceptions and usually you are not able to fix the logic of these functions to fix those exceptions. Therefore, you can only catch and handle these exceptions properly.

All self-defined exceptions are fixed by adding exception handlers to capture and handle the self-defined exceptions that are raised by the application. Although using custom exceptions can make it easy for developers to handle specific use cases and may aid in debugging with descriptive exception names and clear exception messages, developers sometimes may ignore handling custom exceptions as they are app-specific, and thus developers are less familiar with them compared to those commonly used exceptions. Developers should be more careful while using custom exceptions.

4.5 RQ3: Characteristics On Improper Exception Handling Bugs

In this section, we discuss the characteristics of improper exception handling bugs.

4.5.1 Exception Types

For exceptions in improper exception handling bugs, we refer to the exceptions caught in the handler block, i.e., the exception used in the "rescue" statement. The numbers in the parentheses of table 4.3 show the exceptions involved in improper exception handling bugs. These exceptions cover 5 exception types, i.e., 5 exceptions from *Database*, 5 exceptions from *File*, 3 exceptions from *Network Connection*, 1 exception from *Grammar Exceptions*, and 1 exception from *Condition Check Failure Exceptions*. There is no *String Checking Exceptions* in our collected improper exception handling bugs.

Discussions. Our results show that, similar to no exception handling bugs, external-resource-related exceptions are dominant and take an even higher percentage (13 out of 15) and developers should be more careful with the exception handler blocks of external-resource-related exceptions.

Table 4.6: Root causes of improper exception handling bugs.

Root cause	Total
Wrong handler	3
Improper logging	11
Wrong hanlding logic	1

4.5.2 Root Cause and Triggering Conditions

As shown in table 4.6, we categorize improper exception handling bugs into 3 categories, i.e., wrong handler, improper logging, and wrong handling logic.

Wrong handler. Sometimes, although the developers have an exception handler with the intention to catch a certain exception, due to the library being deprecated or developers using a wrong exception type, the handler block fails to handle the intended exception.

Improper logging. Developers may miss logging, not log enough informative messages, or log too much data in handler blocks, which cause other developers to complain in later debugging.

Wrong handling logic. Developers implement wrong handling logic in the exception handler block, and thus leading to unexpected results.

Discussions. Logging is a common operation used in the handler block to collect information to help developers to debug. Therefore, it is important to deliver informative and friendly error messages to developers. 11 out of 15 bugs are about developers complaining that there is no logged information or that logged information is not good enough. Our results suggest that developers should pay more attention to the logged message in the handler block.

4.5.3 Fixing Strategies

We categorize the fixing strategies of the 15 improper exception handling bugs into 5 types. Table 4.7 lists all the five types. They are:

Change exception handler. For the 3 bugs that are caused by using the wrong exception handler type, developers fix them by changing the exception handler type to the correct one.

Make semantic changes. Only 1 bug is fixed by modifying the logic of the code in the handler block so that the exception can be handled as expected.

Add logging. We have 2 bugs that developers add logging in the handler block to gather information to help them debug and fix the bug later.

Refine logging. Although 7 bugs already have loggings in the handler, they do not cover enough key messages to help developers to debug. To fix them, developers add more

Table 4.7: Fixing strategies used for improper exception handling bugs.

Fix	Total
Change handler type	3
Make semantic changes	1
Add logging	2
Refine logging	7
Increase logging level	2

detailed information in the logging statements.

Increase logging level. Log levels are essentially labels indicating the severity of the logged info of your application. The lower the level is, the more information the log will cover, as it will keep track of more unimportant data. We have 2 bugs that developers increase the logging level so that their production logs will not be flooded with unnecessary information.

Discussions. For most improper exception handling bugs, the handling logic in the handler block is usually fine. Developers sometimes will only improve the logging, i.e., adding a log statement, refining the logging statement, or increasing the logging level, to help themselves to better debug exceptions.

4.6 Threats to Validity

In this section, we describe several potential validity threats our study may be subject to and our ways to address them.

(1) The applications in our study cannot represent all real-world web applications. To minimize this threat, we choose popular web applications that are built on top of the popular ORM framework, i.e., Ruby on Rails. These selected web applications are all popular in Github and widely used in previous studies.

(2) We may miss relevant bug reports while searching for keywords for exception handling bugs in the bug tracking system. We mitigate this threat by using keyword search in both bug descriptions and discussion as well as the commit history.

(3) We inspect bug reports manually, which may be subject to human errors while characterizing exception handling bugs. To alleviate this threat, each bug report is examined by at least two people. They all first independently investigate the collected. Once they finish, they cross-check their results and reach an agreement by a group discussion.

4.7 Conclusion

In modern web applications, failing to catch and handle exceptions can lead to exception handling bugs, which can crash the request handler and do much harm to user experience. In this paper, we conduct a comprehensive characteristics study on 214 exception handling bugs collected from 7 real-world database-backed web applications. We examine the exceptions involved in each bug and categorize them into 6 main exception types. We also summarize root causes, triggering conditions, and fixing strategies for these bugs. We expect our results to be useful in guiding future design and implementation of tools for combating exception handling bugs in web applications.

CHAPTER

5

GUIDELINES IN DESIGNING TOOL SUPPORTS

While the results from our studies can help developers understand performance bugs, deadlocks, and exception handling bugs in database-backed web applications, they are more geared toward providing actionable guidance for developing tools that address these issues. This is because our research questions are designed with a focus on aspects like patterns and manifestation conditions, the results of which can be leveraged for tool support. In this chapter, we discuss opportunities in designing tool support for detecting performance bugs, deadlocks, and exception handling bugs based on the results of our research questions. Additionally, we identified opportunities for automated fixes specifically for no-exception-handling bugs, based on our study results.

5.1 Performance Bugs Detection

Our study results on performance bugs can provide guidance in building tools for bug detection, specifically rule-based detection tools. Tool developers can identify statically checkable rules in both ORM-based and raw-SQL web applications and build checkers to detect performance bugs. However, raw-SQL applications may present additional chal-

lenges when implementing static rule checking. Below, we discuss how our results can guide rule-based detection works and unique challenges that need to be addressed while conducting static rule checking for raw-SQL web applications.

The rule-based detection approach is effective for identifying performance bugs, as many of the studied performance issues observed in both ORM-based and raw-SQL web applications have inefficient query patterns, e.g., using less performant queries, in the code. Tool researchers can identify rules that can be statically checked from these patterns, build a static rule checker, and find new instances of violations using these checkers. The previous study on performance bugs in ORM-based web applications conducted such a static analysis with simple rules that can be easily checked using basic regular expression string matching techniques [168] and detected many new instances.

Our study complements the existing studies by conducting a study on performance bugs in raw-SQL web applications. From the studied bugs, we can also identify rules that can be easily checked with string match. For example, using **INTERVAL** is more efficient than **TO_DAYS** when calculating time intervals in a SQL query, or using a single **REPLACE** query is better than using **DELETE** and **INSERT** queries for updating a table. Tool developers can implement static rule checkers that apply string matching to automatically detect those less performant SQL usage patterns in applications' source code.

However, compared with ORM-based web applications, raw-SQL web applications present additional challenges since queries are often dynamically generated, making many rules identified from bugs require more effort to be checked. For example, some queries are redundant and can be skipped because they are repeatedly used with the same results. To detect such bugs, the static checker can look for repeated queries. Since queries in raw-SQL web applications are usually dynamically generated, it is difficult to check if queries are repeated only involving string match. The lack of tools to analyze dynamically generated queries makes rule checking extremely challenging. Although a previous work [161] applied string analysis [76] approach to building a tool to statically check dynamically generated queries, the tool is not publicly available. Therefore, developing and maintaining a tool that can effectively recognize and analyze dynamically generated database queries should be highly valuable, enabling the automated checking of more rules identified from our studied bugs.

5.2 Deadlock Detection

While previous work [87] has utilized static analysis to identify hold-and-wait cycles, we argue that dynamic analysis offers a more scalable and effective solution for modern appli-

cations. Guided by our results on database deadlock patterns, we believe tool developers can build a new database deadlock detection using dynamic analysis, which is more effective in detecting deadlocks in modern web applications. Below, we describe what are the limitations of the existing static approach and how to make use of our study results to build a dynamic deadlock detection tool.

Previous work [87] uses static analysis to identify hold-and-wait cycles and detect database deadlocks. However, the tool proposed in the existing work has the following limitations:

- Firstly, The tool proposed in the previous work relies on manually extracted SQL queries, which are not scalable for complex modern web applications. This is because a modern web application can consist of thousands of SQL queries and many of them are dynamically generated at runtime.
- Secondly, the previous work does not model the "happens-before" relationships and determine whether two transactions can run concurrently. In some scenarios, one request can only be sent after the response of a previous request has been received. The transactions in such two requests cannot be run concurrently.
- Lastly, the previous work has modeled locking behaviors in queries too conservatively, which can only cover a few types of deadlock patterns.

Given the limitations of the existing approach, a dynamic analysis tool informed by key insights from our study would provide a more effective solution for detecting database deadlocks. Our study shows that all the observed deadlocks are triggered by two concurrent transactions, where the interleaving of queries creates a hold-and-wait cycle, leading to a deadlock. We summarized various hold-and-wait cycle patterns in our study, and many of them cannot be modeled by the tool proposed in the previous work [87]. Future developers can build a dynamic deadlock detection tool following the tracing-inference-validation flow. In this architecture, a tool first traces requests during runtime where deadlocks do not manifest. Then, it infers deadlocks by analyzing possible interleavings of queries between two potentially concurrent transactions and identifying circular wait conditions that match the hold-and-wait patterns from our study. Finally, the tool confirms the inferred deadlocks by validation. A key challenge developers may face is the need to model "happens-before" relationships to determine whether two traced requests can run concurrently, which is an aspect not addressed in the previous work [87].

To address the challenges above, we can leverage our previous work ReqRacer [136], which I helped design and develop. To determine whether two requests can run concurrently, ReqRacer constructs a dependency graph to model "happens-before" relationships

between traced requests. If two requests in the dependency graph are not reachable from one another, they are considered capable of running concurrently.

Based on our study results of hold-and-wait cycle patterns, tool developers can implement a dynamic analysis tool to detect deadlocks following a workflow similar to ReqRacer, leveraging its "happens-before" relationship modeling. In the tracing step, we mimic the way how users interact with web applications during normal runs, where deadlocks do not manifest. The tool records certain information during the dynamic executions, e.g., HTTP requests and transactions in each request, and constructs a graph based on the dependencies between requests. This graph is then used to determine whether two transactions can run concurrently. In the inferring step, the tool consumes the traces recorded in the first step to detect deadlocks by inferring potential interleavings of queries between two concurrent transactions, creating a circular wait condition that matches the hold-and-wait cycle patterns identified in our study. Finally, in the validation step, the tool confirms the deadlock by inserting delays between queries in each transaction to enforce the inferred hold-and-wait cycle and then executing the two transactions concurrently to see if the database reports a deadlock error. This tool would help developers effectively detect and resolve database deadlocks in modern web applications.

5.3 Exception Handling Bugs Detection And Fixing

To the best of our knowledge, there are no existing tools on detecting exception handling bugs in database-backed web applications yet. Based on some no exception handling bug code patterns we have seen in our collected bugs and the fixing strategies used to fix them, we believe tool developers can build a tool that first uses rule-based static checkers to detect no exception handling bugs in applications' source code and then automatically generate patches to fix those detected bugs. We will describe how our study results can guide the design and implementation of the tool for automated detection and fixing.

In our collected no exception handling bugs, we observed some APIs are particularly prone to missing exception handling, and tool researchers may be able to identify rules that can be statically checked from these patterns. For example, developers often miss handling no such record exceptions when using the **find()** function to retrieve nonexistent records or miss handling no such file exceptions when using **YAML::load()** to read nonexistent files. In these cases, a general rule can be checking if there is an exception handler for an API that is prone to missing handling a certain exception, and developers can implement static checkers using string match to search for such violations in web applications' source code.

However, only using string match to implement such kind of checkers can easily lead

to false positives, as string match can only check if there is no exception handling in the current function where the API is in, but there may be an exception handler used in an upper-level caller of the current function. Therefore, other static analysis techniques, e.g., control flow analysis, may also be needed to make sure that the tool will detect a true no exception handling bug. Specifically, control flow analysis (CFG) can be employed to inspect all the caller functions from the point where the API is called up to the outermost caller function. This analysis would check each caller in the call hierarchy to determine whether any of them includes proper exception handling for the API in question. By examining the full call stack, the tool can accurately detect whether the exception is unhandled or properly caught at some higher level. This approach ensures that only genuine no exception handling bugs are flagged, thereby reducing false positives.

Once a violation is detected by the static checkers mentioned above, simple patches can be automatically generated to fix the bug. While automated fixing is generally challenging for performance bugs and deadlocks due to the complexity and the potential need for changes in business logic, we found that the fixing strategies for these specific misused APIs are typically straightforward. In these cases, the solution usually involves wrapping the vulnerable APIs in a try-catch block to handle the expected exceptions and return appropriate error messages to the user. Since these particular APIs tend to require only simple exception handling, the tool could automatically insert a try-catch block around the misused API without the need for more complex logic or customization. This makes automated fixing feasible and efficient for these scenarios, unlike more intricate bugs that may involve deeper changes in application logic.

CHAPTER

6

RELATED WORK

In this chapter, we are going to introduce some existing work that are related to our topic.

6.1 Performance Bugs and Antipatterns

In Section 2.1 of Chapter 2, we have discussed some related work on database-access antipatterns and performance-bug studies. We are next going to introduce some other work.

Various previous approaches focus on detecting antipatterns. Some of them do not focus on performance antipatterns but on functionality antipatterns [52, 70, 99], and some of them focus on performance antipatterns in ORM-based applications [68, 71, 169]. Our recent work [150] is unique in that we target at reporting all known database-access performance antipatterns that can be found in the literature and complementing existing studies by focusing on raw-SQL web application and our results suggest that there are multiple performance antipatterns that are currently missed by existing work on ORM-based web applications, raw-SQL web applications deserve more research attention.

Both static approaches [71, 92, 113, 128, 148, 163, 166, 169] and dynamic approaches [127, 129, 162] have been explored to detect different types of performance issues. It is challenging to develop static checkers for applications studied in our work, as currently there is a

lack of mature implementations that can statically analyze dynamically generated queries involving multiple programming languages.

There are multiple pieces of work focusing on improving the performance of database-backed applications [59, 69, 72, 73, 120, 139]. Although some of the key ideas are similar to the fix strategies seen in performance-bug patches, such as reducing database accesses and optimizing query execution efficiency, these database-side optimization approaches cannot optimize away the performance inefficiencies of related bugs in our study [150], as existing approaches are usually limited to a single round-trip between application and database, while many database-access antipatterns involve multiple round-trips.

6.2 Concurrency Bugs

In Section 3.1 of Chapter 3, we have discussed some related work on deadlocks in multi-threaded programs and web applications. The results of our deadlock study [135] suggest that existing work cannot handle a large portion of real-world deadlocks in web applications, especially those inter-request deadlocks on database locks. While there are studies focusing on concurrency bugs in web applications [78, 136, 137, 156], they do not cover deadlocks.

A lot of research efforts have been spent on races and concurrency bugs in multi-threaded programs. Researchers have conducted thorough empirical characteristic studies [84, 115] and the study results guide the development of tools for various purposes, e.g., bug detection [57, 58, 83, 90, 116, 146, 174, 175], program testing [89, 106, 131, 149, 155, 171], failure diagnosis [50, 51, 94, 100, 117], and fixing [93, 95, 111, 112]. Researchers have also proposed techniques targeting process races on the operating-system level [103] and distributed concurrency bugs in distributed and cloud systems [104, 105, 109, 110, 114].

Moreover, some other recent work also focuses on client-side race detection [44, 56, 91, 122, 132, 141, 173] and fixing [43].

6.3 Exception Handling Bugs

Some existing studies have been conducted on error handling used in operating system (OS) kernels and file systems [88, 130, 142]. However, OS kernels and file systems are implemented in C language while web applications are usually implemented in Object-Oriented (OO) programming languages, such as PHP, Java, and Ruby. C language is different from these OO programming languages as it does not provide structured error handling primitives, i.e., C language does not have a throw-catch-handle mechanism as the OO

programming language does to handle exceptions. C language handles exceptions by just checking the returned exception code, which is usually an integer value to indicate success or failure. In other words, the gained insights of exception handling used in OS cannot be generalized to EH bugs in database-backed web applications.

Yuan et al. [172] studied catastrophic failures reported in distributed systems that are implemented in Java. As an OO programming language, Java uses the language-provided exception handling mechanism to separate the functional code from exception handling logic. Their paper found that some catastrophic failures are triggered by wrong error handling logic implemented in the exception handler block. However, the failures that they studied are from distributed systems, not web applications, and they did not mention if there are failures that are caused by missing handling of certain exceptions in their paper. It's not clear what the characteristics of missing exception handling and improper exception handling bugs are in database-backed web applications.

6.4 Server-side Web Applications

Server-side web applications also have been the subject of a lot of existing research, and we next briefly discuss other related work on server-side web applications. Many different techniques have been proposed for improving their reliability [48, 49, 54, 80, 85, 124, 125, 126, 145], mostly focusing on program analysis, bug detection, input generation, or automated repair.

Techniques focusing on the security aspect of web applications have also been proposed, e.g., auditing [101, 153], intrusion detection and recovery [63, 64, 121], identifying information disclosure [67]. However, none of the work mentioned above handles performance bugs, deadlocks, and exception handling bugs.

CHAPTER

7

CONCLUSION AND FUTURE WORK

In this chapter, we will first conclude our work presented in Chapter 2, Chapter 3, and Chapter 4. After that, we will discuss potential next steps based on the current progress and results of our studies.

7.1 Conclusion

This thesis focuses on aiding developers in understanding and combating performance and reliability issues in database-backed web applications. This is achieved through comprehensive empirical studies on real-world performance bugs, deadlocks, and exception handling bugs collected from popular open-source applications.

On performance, we have present a comprehensive empirical study that characterizes performance antipatterns related to database accesses in web applications. From our literature survey, we have summarize and report a total of 24 known performance antipatterns. The comprehensiveness of our results makes them a valuable reference for future work on database-access performance antipatterns. Additionally, based on real-world performance bugs from direct-accessing web applications, we identify 10 new database-access performance antipatterns that were not previously reported in the existing research literature.

On deadlocks, we characterize deadlocks from real-world web applications based on

the number of HTTP requests and the types of resources involved. Our results suggest that deadlocks on database locks are not only the most common but also the most challenging type of deadlocks in web applications, which are worth further investigation. Based on the collected real-world database-lock deadlocks, along with additional StackOverflow question posts, we summarize four hold-and-wait cycle patterns of deadlocks on database locks. Additionally, we also categorize fixing strategies for these real-world deadlocks.

On exception handling, we conduct a comprehensive characteristic study on 214 exception handling bugs collected from 7 real-world database-backed web applications. This bug set includes 199 bugs that are caused by failing to catch certain exceptions and 15 bugs that are caused by handling exceptions improperly. We examine the exceptions involved in each bug and categorize them into six main exception types. We also summarize root causes, triggering conditions, and fixing strategies for these bugs. We expect our results to be useful in guiding future design and implementation of tools for combating exception handling bugs in web applications.

7.2 Future Work

With our experience in conducting empirical studies on characterizing real-world bugs in database-backed web applications, and based on the derived characteristic results, we foresee a few potential future directions as follows:

Performance and concurrency. Given the conccurnet nature of web applications, developers must take potential concurrency issues into consideration during implementation. However, they sometimes could be overthinking and take unnecessary synchronization operations, which leads to deadlocks instead and thus hurts the performance. Our deadlock study presented in Chapter 3 can confirm this observation as we see deadlocks that are fixed by removing unnecessary locks and transactions. Currently, the performance bugs studied in Chapter 2 are not related to concurrency issues. Therefore, we can study real-world bugs that are related to both concurrency and performance to characterize the relationhip between concurrency and performance. This can help in removing unnecessary synchronization operations and improving performance.

Understanding performance and reliability issues in other systems. My thesis has focused on web applications built with an SQL-based database. However, similar performance and reliability problems are also likely to exist in similar systems, such as mobile applications or applications that use non-SQL databases. My future research goal is to continue to help developers better understand performance and reliability problems in these applications by conducting studies on real-world bugs collected from such applications.

Database-lock deadlock detection tool. Our deadlock study in Chapter 3 has identified some hold-and-wait cycle patterns that previous work did not cover, thus there are no existing techniques that can detect such deadlocks. Based on these patterns, we will build tools to detect potential deadlocks. More specifically, we plan to instrument web applications to trace queries executed in each request while doing normal in-house testing. Then we will developer checkers to determine if the logged queries from two concurrent requests match our summarized hold-and-wait cycle patterns.

Automated testing and fixing framework for exception handling bugs. In Chapter 4, we find that many exception handling bugs are related to empty input values or null values, and their fixes often involve adding an if condition to prevent these invalid inputs. Therefore, we propose building a framework to automatically test and fix such bugs. This framework will automatically generate empty and null values for triggering no exception handling bugs. If such an exception is found, then the framework will automatically add an if condition check as a patch to preclude the invalid input case.

REFERENCES

[1] [n.d.]. 15.7.3 Locks Set by Different SQL Statements in InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-locks-set.html.

[2] [n.d.]. 20 Most Famous Web Apps Built with Ruby on Rails. https://softcircles.com/blog/20-most-famous-web-apps-built-with-ruby-on-rails.

[3] [n.d.]. A puzzled MySQL deadlock caused by two update..in. https://stackoverflow.com/questions/65519414.

[4] [n.d.]. An insert caused a deadlock in InnoDB. How did this happen? https://stackoverflow.com/questions/24327317.

[5] [n.d.]. Avoiding MySQL deadlock when upgrading shared to exclusive lock. https://stackoverflow.com/questions/41015813.

[6] [n.d.]. BugZilla. https://bugzilla.mozilla.org/.

[7] [n.d.]. DB deadlock if user edits issue when project is being analyzed. https://jira.sonarsource.com/browse/SONAR-11097.

[8] [n.d.]. DBError 'Error: 1213 Deadlock found when trying to get lock' on WikiPage::doUpdateRestrictions. https://phabricator.wikimedia.org/T214035.

[9] [n.d.]. Deadlock INSERT IGNORE INTO `flaggedtemplates`. https://phabricator.wikimedia.org/T30598.

[10] [n.d.]. Deadlock on cache_config (DatabaseBackend::setMultiple()). https://www.drupal.org/project/drupal/issues/2336627.

[11] [n.d.]. Deadlock on user account creation leaves account in bad state. https://phabricator.wikimedia.org/T38116.

[12] [n.d.]. Deadlocks - PostgreSQL Documentation. https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-DEADLOCKS.

[13] [n.d.]. Deadlocks as a result of the speedup by joining on idset table. https://issues.openmrs.org/browse/REPORT-674.

[14] [n.d.]. deadlocks on INSERT IGNORE INTO wbc_entity_usage. https://phabricator.wikimedia.org/T192349.

[15] [n.d.]. Detecting and Ending Deadlocks - SQL Server technical documentation. https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15#detecting-and-ending-deadlocks.

[16] [n.d.]. Diaspora. https://github.com/diaspora/diaspora.

[17] [n.d.]. DNN Platform Issue Tracker. https://dnntracker.atlassian.net.

[18] [n.d.]. DRD: a thread error detector. https://www.valgrind.org/docs/manual/drd-manual.html.

[19] [n.d.]. Find your new favorite web framework. https://hotframeworks.com.

[20] [n.d.]. Helgrind: a thread error detector. https://www.valgrind.org/docs/manual/hg-manual.html.

[21] [n.d.]. Innodb update locking. https://stackoverflow.com/questions/40653848.

[22] [n.d.]. Joomla! Developer Network. https://developer.joomla.org/.

[23] [n.d.]. Lobsters. https://github.com/lobsters/lobsters.

[24] [n.d.]. mariadb deadlocked occasionally. https://github.com/lobsters/lobsters-ansible/issues/39.

[25] [n.d.]. Moodle Tracker. https://tracker.moodle.org/.

[26] [n.d.]. mysql CREATE temporary table + Transaction causes deadlock. https://stackoverflow.com/questions/23768456.

[27] [n.d.]. MySQL Deadlock Detection - MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-detection.html.

[28] [n.d.]. Mysql deadlock explanation needed. https://stackoverflow.com/questions/1851528.

[29] [n.d.]. MySQL return Deadlock with insert row and FK is locked 'for update'. https://stackoverflow.com/questions/2560070.

[30] [n.d.]. mysql transaction deadlock. https://stackoverflow.com/questions/5353877.

[31] [n.d.]. Odoo Issues. https://github.com/odoo/odoo/issues/.

[32] [n.d.]. Onebody. https://github.com/seven1m/onebody.

[33] [n.d.]. OpenStreetMap. https://github.com/openstreetmap/openstreetmap-website.

[34] [n.d.]. Redmine. https://www.redmine.org/.

[35] [n.d.]. Spree. https://github.com/spree/spree.

[36] [n.d.]. The RedMonk Programming Language Rankings: January 2023. https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/.

[37] [n.d.]. Top 15 Most Popular Websites | January 2020. http://www.ebizmba.com/articles/most-popular-websites.

[38] [n.d.]. Tracks. https://github.com/TracksApp/tracks.

[39] [n.d.]. Usage statistics and market share of WordPress for websites. https://w3techs.com/technologies/details/cm-wordpress/all/all.

[40] [n.d.]. Why MySQL InnoDB creates so many deadlocks when Hangfire enques multiple jobs in parallel? https://stackoverflow.com/questions/53854450/.

[41] [n.d.]. Wikimedia Phabricator. https://phabricator.wikimedia.org/.

[42] [n.d.]. WordPress Trac. https://core.trac.wordpress.org/.

[43] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 289–299. https://doi.org/10.1109/ICSE.2017.34

[44] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical Initialization Race Detection for JavaScript Web Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 66 (Oct. 2017), 22 pages. https://doi.org/10.1145/3133890

[45] Rahul Agarwal and Scott D. Stoller. 2006. Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD) (PADTAD '06)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/1147403.1147413

[46] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2005. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing (HVC) (HVC'05)*. Springer-Verlag, Berlin, Heidelberg, 191–207. https://doi.org/10.1007/11678779_14

[47] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2901739.2901774

[48] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Practical fault localization for dynamic web applications. In *2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, Vol. 1. 265–274. https://doi.org/10.1145/1806799.1806840

[49] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering* 36, 4 (2010), 474–494. https://doi.org/10.1109/TSE.2010.31

[50] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2451116.2451128

[51] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages*

and Operating Systems (ASPLOS '14). ACM, New York, NY, USA, 207–222. https://doi.org/10.1145/2541940.2541973

[52] N. Arzamasova, M. Schäler, and K. Böhm. 2018. Cleaning Antipatterns in an SQL Query Log. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (March 2018), 421–434. https://doi.org/10.1109/TKDE.2017.2772252

[53] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. 2020. On the Prevalence, Impact, and Evolution of SQL code smells in Data-Intensive Systems. In *[Provisoire] Proceedings of the 17th International Conference on Mining Software Repositories (MSR 2020)*. ACM Press, United States.

[54] Snigdha Athaiya and Raghavan Komondoor. 2017. Testing and Analysis of Web Applications Using Page Models. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (ISSTA 2017)*. ACM, New York, NY, USA, 181–191. https://doi.org/10.1145/3092703.3092734

[55] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1327–1342. https://doi.org/10.1145/2723372.2737784

[56] Marina Billes, Anders Møller, and Michael Pradel. 2017. Systematic Black-box Analysis of Collaborative Web Applications. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 171–184. https://doi.org/10.1145/3062341.3062364

[57] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 28–39. https://doi.org/10.1145/2594291.2594323

[58] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[59] Ivan T. Bowman and Kenneth Salem. 2005. Optimization of Query Streams Using Semantic Prefetching. *ACM Trans. Database Syst.* 30, 4 (Dec. 2005), 1056–1101. https://doi.org/10.1145/1114244.1114250

[60] Yan Cai and Lingwei Cao. 2016. Fixing Deadlocks via Lock Pre-Acquisitions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1109–1120. https://doi.org/10.1145/2884781.2884819

[61] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *2012 34th International Conference on Software Engineering (ICSE)*. 606–616. https://doi.org/10.1109/ICSE.2012.6227156

[62] Yan Cai, Shangru Wu, and W. K. Chan. 2014. ConLock: A Constraint-Based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE) (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 491–502. https://doi.org/10.1145/2568225.2568312

[63] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. 2011. Intrusion Recovery for Database-backed Web Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP) (SOSP '11)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/2043556.2043567

[64] Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. 2013. Asynchronous Intrusion Recovery for Interconnected Web Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP) (SOSP '13)*. ACM, New York, NY, USA, 213–227. https://doi.org/10.1145/2517349.2522725

[65] Surajit Chaudhuri, Vivek Narasayya, and Manoj Syamala. 2007. Bridging the Application and DBMS Profiling Divide for Database Application Developers. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 1252–1262.

[66] Boyuan Chen, Zhen Ming (Jack) Jiang, Paul Matos, and Michael Lacaria. 2019. An Industrial Experience Report on Performance-Aware Refactoring on a Database-Centric Web Application. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE Press, 653–664. https://doi.org/10.1109/ASE.2019.00066

[67] Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2014. Identifying Information Disclosure in Web Applications with Retroactive Auditing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI) (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 555–569. http://dl.acm.org/citation.cfm?id=2685048.2685092

[68] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Transactions on Software Engineering* 42, 12 (Dec 2016), 1148–1161. https://doi.org/10.1109/TSE.2016.2553039

[69] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-Based Database-Centric Web Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 666–677. https://doi.org/10.1145/2950290.2950303

[70] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. Detecting Problems in the Database Access Code of Large Scale Systems: An Industrial Experience Report. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/2889160.2889228

[71] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1001–1012. https://doi.org/10.1145/2568225.2568259

[72] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *Proc. VLDB Endow.* 5, 11 (July 2012), 1471–1482. https://doi.org/10.14778/2350229.2350262

[73] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014. Sloth: Being Lazy is a Virtue (when Issuing Database Queries). In *Proceedings of the 2014 ACM SIGMOD*

International Conference on Management of Data (SIGMOD '14). ACM, New York, NY, USA, 931–942. https://doi.org/10.1145/2588555.2593672

[74] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C Myers. 2014. Using Program Analysis to Improve Database Applications. *IEEE Data Engineering Bulletin* 37, 1 (2014), 48–59.

[75] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

[76] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis (SAS '03)*. Springer-Verlag, Berlin, Heidelberg, 1–18. http://dl.acm.org/citation.cfm?id=1760267.1760269

[77] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2015. Dynamic Deadlock Verification for General Barrier Synchronisation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 150–160. https://doi.org/10.1145/2688500.2688519

[78] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 145–160. https://doi.org/10.1145/3064176.3064188

[79] Robert F. Dugan, Ephraim P. Glinert, and Ali Shokoufandeh. 2002. The Sisyphus Database Retrieval Software Performance Antipattern. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP '02)*. Association for Computing Machinery, New York, NY, USA, 10–16. https://doi.org/10.1145/584369.584372

[80] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic Test Input Generation for Database Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA) (ISSTA '07)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1273463.1273484

[81] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP) (SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[82] Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: Scalable Deadlock Detection for Concurrent Programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 353–365. https://doi.org/10.1145/2635868.2635918

[83] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[84] P. Fonseca, Cheng Li, V. Singhal, and R. Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 221–230. https://doi.org/10.1109/DSN.2010.5544315

[85] Maryam Abdul Ghafoor, Muhammad Suleman Mahmood, and Junaid Haroon Siddiqui. 2016. Effective Partial Order Reduction in Model Checking Database Applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 146–156. https://doi.org/10.1109/ICST.2016.25

[86] Mark Grechanik, B. M. Mainul Hossain, and Ugo Buy. 2013. Testing Database-Centric Applications for Causes of Database Deadlocks. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, USA, 174–183. https://doi.org/10.1109/ICST.2013.19

[87] Mark Grechanik, B. M. Mainul Hossain, Ugo Buy, and Haisheng Wang. 2013. Preventing Database Deadlocks in Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 356–366. https://doi.org/10.1145/2491411.2491412

[88] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In

Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08). USENIX Association, USA, Article 14, 16 pages.

[89] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 210–220. https://doi.org/10.1145/2338965.2336779

[90] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[91] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless Model Checking of Event-driven Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 57–73. https://doi.org/10.1145/2814270.2814282

[92] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[93] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[94] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 241–255. https://doi.org/10.1145/1869459.1869481

[95] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating*

Systems Design and Implementation (OSDI'12). USENIX Association, Berkeley, CA, USA, 221–236. http://dl.acm.org/citation.cfm?id=2387880.2387902

[96] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. 2010. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) (FSE'10)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/1882291.1882339

[97] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/1542476.1542489

[98] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems to Defend against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI) (OSDI'08)*. USENIX Association, USA, 295–308.

[99] Bill Karwin. 2010. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming* (1st ed.). Pragmatic Bookshelf.

[100] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

[101] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2012. Efficient Patch-based Auditing for Web Application Vulnerabilities. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI) (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 193–206. http://dl.acm.org/citation.cfm?id=2387880.2387899

[102] Barbara Kitchenham. 2004. *Procedures for Performing Systematic Reviews*. Joint Technical Report, Computer Science Department, 2004, Keele University (TR/SE-0401) and National ICT Australia Ltd. (0400011T.1).

[103] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. 2011. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 353–367. https://doi.org/10.1145/2043556.2043589

[104] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[105] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 994–1009. https://doi.org/10.1145/3314221.3314620

[106] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-safety-violation Detection: Finding Thousands of Concurrency Bugs During Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[107] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. 2005. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC) (ATEC '05)*. USENIX Association, USA, 31–44.

[108] Yiyan Lin and Sandeep S. Kulkarni. 2014. Automatic Repair for Multi-Threaded Programs with Deadlock/Livelock Using Maximum Satisfiability. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/2610384.2610398

[109] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*

'17). ACM, New York, NY, USA, 677–691. https://doi.org/10.1145/3037697.3037735

[110] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 419–431. https://doi.org/10.1145/3173162.3177161

[111] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: Context-aware Fixing of Concurrency Bugs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 318–329. https://doi.org/10.1145/2635868.2635881

[112] Peng Liu and Charles Zhang. 2012. Axis: Automatically Fixing Atomicity Violations Through Solving Control Constraints. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 299–309. http://dl.acm.org/citation.cfm?id=2337223.2337259

[113] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[114] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-Mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/3236024.3236071

[115] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[116] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 277–288. https://doi.org/10.1109/ISCA.2008.4

[117] Brandon Lucia, Benjamin P. Wood, and Luis Ceze. 2011. Isolating and Understanding Concurrency Errors Using Reconstructed Execution Fragments. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 378–388. https://doi.org/10.1145/1993498.1993543

[118] Y. Lyu, A. Alotaibi, and W. G. J. Halfond. 2019. Quantifying the Performance Impact of SQL Antipatterns on Mobile Applications. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 53–64.

[119] Yingjun Lyu, Ding Li, and William G. J. Halfond. 2018. Remove RATs from Your Code: Automated Optimization of Resource Inefficient Database Writes for Mobile Applications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 310–321. https://doi.org/10.1145/3213846.3213865

[120] Amit Manjhi, Charles Garrod, Bruce M. Maggs, Todd C. Mowry, and Anthony Tomasic. 2009. Holistic Query Transformations for Dynamic Web Applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. IEEE Computer Society, Washington, DC, USA, 1175–1178. https://doi.org/10.1109/ICDE.2009.194

[121] David R. Matos, Miguel L. Pardal, and Miguel Correia. 2017. Rectify: Black-box Intrusion Recovery in PaaS Clouds. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware) (Middleware '17)*. ACM, New York, NY, USA, 209–221. https://doi.org/10.1145/3135974.3135978

[122] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 381–392. https://doi.org/10.1145/2786805.2786820

[123] Csaba Nagy and Anthony Cleve. 2017. A static code smell detector for SQL queries embedded in java code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 147–152.

[124] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2013. Dangling references in multi-configuration and dynamic PHP-based Web applications. In *2013 28th IEEE/ACM International Conference on*

Automated Software Engineering (ASE). 399–409. https://doi.org/10.1109/ASE.2013.6693098

[125] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2011. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 13–22. https://doi.org/10.1109/ASE.2011.6100047

[126] Hung Viet Nguyen, Hung Dang Phan, Christian Kästner, and Tien N. Nguyen. 2019. Exploring Output-based Coverage for Testing PHP Web Applications. *Automated Software Engg.* 26, 1 (March 2019), 59–85. https://doi.org/10.1007/s10515-018-0246-5

[127] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 268–278. https://doi.org/10.1145/2491411.2491416

[128] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 237–246. http://dl.acm.org/citation.cfm?id=2487085.2487134

[129] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 562–571. http://dl.acm.org/citation.cfm?id=2486788.2486862

[130] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1203–1218. https://doi.org/10.1145/3372297.3417256

[131] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[132] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 251–262. https://doi.org/10.1145/2254064.2254095

[133] Hari K. Pyla and Srinidhi Varadarajan. 2010. Avoiding deadlock avoidance. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 75–85. https://doi.org/10.1145/1854273.1854288

[134] Dong Qiu, Bixin Li, and Zhendong Su. 2013. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, New York, NY, USA, 125–135. https://doi.org/10.1145/2491411.2491431

[135] Zhengyi Qiu, Shudi Shao, Qi Zhao, and Guoliang Jin. 2021. A Characteristic Study of Deadlocks in Database-Backed Web Applications. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Wuhan, China, 510–521. https://doi.org/10.1109/ISSRE52982.2021.00059

[136] Zhengyi Qiu, Shudi Shao, Qi Zhao, and Guoliang Jin. 2021. Understanding and Detecting Server-Side Request Races in Web Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 842–854. https://doi.org/10.1145/3468264.3468594

[137] Zhengyi Qiu, Shudi Shao, Qi Zhao, Hassan Ali Khan, Xinning Hui, and Guoliang Jin. 2022. A Deep Study of the Effects and Fixes of Server-Side Request Races in Web Applications. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. Association for Computing Machinery, New York, NY, USA, 744–756. https://doi.org/10.1145/3524842.3528463

[138] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. 2015. Program Transformations for Asynchronous and Batched Query Submission. *IEEE Transactions on Knowledge and Data Engineering* 27, 2 (2015), 531–544.

[139] Karthik Ramachandra and S. Sudarshan. 2012. Holistic Optimization by Prefetching Query Results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2213836.2213852

[140] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). McGraw-Hill, Inc., USA.

[141] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[142] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

[143] Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA) (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 473–489. https://doi.org/10.1145/2660193.2660238

[144] Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2555243.2555262

[145] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *2012 34th International Conference on Software Engineering (ICSE)*. 277–287. https://doi.org/10.1109/ICSE.2012.6227186

[146] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[147] Ziv Scully and Adam Chlipala. 2017. A Program Optimization for Automatic Database Result Caching. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles*

of Programming Languages (POPL '17). Association for Computing Machinery, New York, NY, USA, 271–284. https://doi.org/10.1145/3009837.3009891

[148] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2884781.2884829

[149] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

[150] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. 2020. Database-Access Performance Antipatterns in Database-Backed Web Applications. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 58–69. https://doi.org/10.1109/ICSME46990.2020.00016

[151] Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. 2018. Smelly Relations: Measuring and Understanding Database Schema Quality. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/3183519.3183529

[152] Juan M. Tamayo, Alex Aiken, Nathan Bronson, and Mooly Sagiv. 2012. Understanding the Behavior of Database Operations under Program Control. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 983–996. https://doi.org/10.1145/2384616.2384688

[153] Cheng Tan, Lingfan Yu, Joshua B. Leners, and Michael Walfish. 2017. The Efficient Server Audit Problem, Deduplicated Re-execution, and the Web. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP) (SOSP '17)*. ACM, New York, NY, USA, 546–564. https://doi.org/10.1145/3132747.3132760

[154] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 4–18. https://doi.org/10.1145/3514221.3526120

[155] Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage Guided Systematic Concurrency Testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/1985793.1985824

[156] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 520–531. https://doi.org/10.1109/ASE.2017.8115663

[157] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 520–531.

[158] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. 2008. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI) (OSDI'08)*. USENIX Association, USA, 281–294.

[159] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. 2009. The Theory of Deadlock Avoidance via Discrete Control. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (POPL '09)*. Association for Computing Machinery, New York, NY, USA, 252–263. https://doi.org/10.1145/1480881.1480913

[160] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037

[161] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. 2007. Static Checking of Dynamically Generated Queries in Database Applications. *ACM Trans. Softw. Eng. Methodol.* 16, 4, Article 14 (Sept. 2007). https://doi.org/10.1145/1276933.1276935

[162] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *Proceed-*

ings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013). ACM, New York, NY, USA, 90–100. https://doi.org/10.1145/2483760.2483784

[163] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, New York, NY, USA, 1299–1308. https://doi.org/10.1145/3132847.3132954

[164] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2020. View-Driven Optimization of Database-Backed Web Applications. In *10th BiAnnualennial Conference on Innovative Data Systems Research (CIDR '20)*. www.cidrdb.org, 7.

[165] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing data constraints in database-backed web applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1098–1109. https://doi.org/10.1145/3377811.3380375

[166] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[167] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[168] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[169] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. PowerStation: Automatically Detecting and Fixing Inefficiencies of Database-Backed

Web Applications in IDE. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 884–887. https://doi.org/10.1145/3236024.3264589

[170] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. 2019. View-Centric Performance Optimization for Database-Backed Web Applications. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 994–1004. https://doi.org/10.1109/ICSE.2019.00104

[171] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. 2014. SimRT: An Automated Framework to Support Regression Testing for Data Races. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/2568225.2568294

[172] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 249–265.

[173] Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 278–288. https://doi.org/10.1109/ICSE.2017.33

[174] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 251–264. https://doi.org/10.1145/1950365.1950395

[175] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 179–192. https://doi.org/10.1145/1736020.1736041

[176] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proceedings*

of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07). Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/1247480.1247540

[177] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. 2017. UNDEAD: Detecting and preventing deadlocks in production software. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 729–740. https://doi.org/10.1109/ASE.2017.8115684