Understanding and Locating Quality Issues in the Database Access Code of Database-Backed Applications

Wei Liu

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

September 2024

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Wei Liu**

Entitled: **Understanding and Locating Quality Issues in the Database Access Code of Database-Backed Applications**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Andrea Schiffauerova

_____ External Examiner
Dr. Cor-Paul Bezemer

_____ Examiner
Dr. Weiyi Shang

_____ Examiner
Dr. Joey Paquet

_____ Examiner
Dr. Shin Hwei Tan

_____ Supervisor
Dr. Tse-Hsun (Peter) Chen

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

18 November 2024 _____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Understanding and Locating Quality Issues in the Database Access Code of Database-Backed Applications

Wei Liu, Ph.D.

Concordia University, 2024

Database-backed applications interact with the database management system (DBMS), such as MySQL, for persistent data storage. These database accesses play a central role in such applications and are crucial for their maintenance and quality. Developers build database-backed applications to access relational databases using object-oriented programming languages such as Java, Python, C#, PHP, and C++. Since object-oriented programming is a different paradigm compared to relational databases, developers use various technologies to ease database access by abstracting persistent data as objects. Specifically, developers often rely on two main access technologies: (i) executing a Structured Query Language (SQL) query and manually converting the results to objects; and (ii) using Object-Relational Mapping (ORM) frameworks, which automatically generate SQL queries and convert the results to objects based on various object-database mapping configurations. However, developers may face different database access challenges when using different technologies. Moreover, due to the abstraction of ORM frameworks, developers may face challenges when debugging database access problems. ORM automatically generates SQL queries based on various ORM configurations (e.g., the relationship among object types) and the invoked ORM APIs. As a result, developers do not have direct control over how ORM generates SQL queries. If there is a database access issue associated with a problematic-generated SQL query, developers may have difficulties knowing how and where the SQL query is generated in the application code, causing challenges in debugging database access problems.

Motivated by the importance and challenges of database access, in this thesis, we first conduct an empirical study of database access bugs in seven large-scale Java open-source applications that use relational database management systems. Specifically, by manually examining the bug reports and commit histories ranging from 5 to 16 years, we investigate and derive the characteristics such as categories, root cause, impact, and occurrence of database access issues when using popular database access technologies. Our empirical study provides motivations and guidelines for future research to help avoid, detect, and test database access bugs in database-backed applications. To

assist developers in debugging database access problems, we propose an approach for locating the origin (i.e., the control flow path containing a sequence of method calls) that generates a given SQL query. It achieves state-of-the-art localization accuracy and improves Top@5 accuracy by 225% and 333% compared to the baseline approach when using SQL session logs and individual query logs, respectively. We also find that our approach can help developers locate data access issues that generate problematic SQL queries (i.e., slow SQL queries and database deadlocks). In conclusion, this thesis uncovers the root causes of database access issues and demonstrates that leveraging both static analysis and information retrieval techniques can help developers debug database access issues associated with problematic SQL queries. It also paves the way for future research on the development and automatic generation of tests for database access code to improve the quality of database-backed applications.

Statement of Originality

I hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. I understand that my thesis may be made electronically available to the public.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Tse-Hsun (Peter) Chen, for his exceptional guidance, patience, insights, and encouragement. Without your supervision and invaluable support, this thesis would not have been possible.

Special thanks to my thesis examiners, Dr. Paquet, Dr. Tan, Dr. Shang, and Dr. Bezemer for their extremely valuable and constructive suggestions.

I would also like to express my sincere gratitude to all the members of the SPEAR lab. Studying in the lab has been a wonderful experience and a rewarding research experience.

Lastly, I want to thank my family and friends for their constant support and understanding during this period. Your love and encouragement have been my primary motivation.

Related Publications

In all chapters and directly related publications of this thesis, my contributions include: drafting the initial research idea, researching background knowledge, reviewing related work, collecting experimental data, implementing tools, conducting experiments, and writing and polishing the manuscript. My co-authors supported me by refining the initial ideas, identifying missing related work, independently deriving a second set of bug causes, providing feedback on earlier drafts, and further polishing the writing.

The following publications are directly related to the materials presented in this thesis:

Wei Liu, Shouvick Mondal, and Tse-Hsun (Peter) Chen, "An Empirical Study on the Characteristics of Database Access Bugs in Java Applications", ACM Transactions on Software Engineering and Methodology (TOSEM), 33, 7, Article 181 (September 2024), 25 pages, doi:10.1145/3672449. This work is discussed in Chapter 3.

Wei Liu, and Tse-Hsun (Peter) Chen, "SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications", IEEE Transactions on Software Engineering (TSE), vol. 49, no. 6, pp. 3376-3390, 1 June 2023, doi: 10.1109/TSE.2023.3253700. This work is discussed in Chapter 4.

The following publications are not directly related to this thesis but were conducted as parallel work to the research presented in this thesis.

Steven Locke, Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and **Wei Liu**, "LogAssist: Assisting Log Analysis Through Log Summarization", IEEE Transactions on Software Engineering (TSE), vol. 48, no. 9, pp. 3227-3241, 1 Sept. 2022, doi: 10.1109/TSE.2021.3083715.

Zehao Wang, **Wei Liu**, Jinfu Chen, and Tse-Hsun (Peter) Chen, "RPerf: Mining User Reviews Using Topic Modeling to Assist Performance Testing: An Industrial Experience Report", Journal of Systems and Software (JSS), under review.

Contents

List of Figures

List of Tables

Part I

Introduction, Background, and Literature Review

Chapter 1

Introduction

1.1 Introduction

Database-backed applications are essential in many areas such as online shopping, social media, banking, and health care. These applications rely on the underlying database management system (DBMS), such as MySQL, for persistent data storage. Specifically, they interact with the database to perform various operations such as retrieving, inserting, updating, or deleting data, which is known as database access. However, the inherent difference between database-backed applications and the underlying DBMS makes database access challenging. In database-backed applications, developers often use object-oriented programming languages such as Java, Python, C#, PHP, and C++ [101, 122] to implement the object and business logic. In relational databases, data is organized in tables with rows and columns, defined by the database schema. Since object-oriented programming is a different paradigm compared to relational databases, developers use two main database access technologies to ease database access: (i) manually constructing raw Structured Query Language (SQL) queries directly and converting the query results to objects; and (ii) using Object-Relational Mapping (ORM) frameworks, which automatically generates SQL queries and converts the query results to objects based on various object-database mapping configurations.

Since database data is critical to database-backed applications, accessing the database correctly and efficiently is crucial. However, developers may face various challenges in accessing the database when using different technologies. Database access issues refer to problems or failures that arise during interactions between applications and databases through SQL queries or ORM frameworks. These issues can manifest as runtime errors (e.g., crashes or exceptions), performance issues (e.g., slow queries or database deadlocks), or incorrect query results caused by bugs in application code, database schema, or ORM configurations. When using raw SQL queries, developers must carefully

construct complex SQL queries that are free from syntax errors and satisfy all database schema constraints. They also need to manually implement code to convert the query results into objects in the applications. On the other hand, when using ORM frameworks, developers may unintentionally misuse the ORM APIs because these frameworks hide the underlying SQL query generation and execution. For instance, calling certain ORM APIs might lead to the generation of SQL queries that eagerly load associated objects from the database, even if those objects are not used in the application (i.e., inefficient eager loading [128]), which can lead to performance issues. In addition, the co-evolution of database schema and application code [125] also makes database access challenging. For instance, developers may be unaware of changes to table column names in databases when updating the corresponding code in applications. Thus, accessing the database may cause runtime exceptions when the column specified in the SQL query does not exist in the database. Due to the importance and challenge of database access, many prior works have studied the maintenance issues of database-backed applications from the perspective of syntactic or semantic errors in SQL queries [28, 17], SQL anti-patterns [77, 24, 52, 18], SQL code smells [126, 129, 100], and performance issues [38, 150, 151, 128]. However, they primarily focus on SQL statements within applications, overlooking database access bugs that may occur during interactions with the database. There is still limited research examining database access bugs arising from the use of SQL queries or ORM frameworks, while considering both database access and business logic code simultaneously.

Using ORM frameworks also presents unique challenges compared to directly using SQL queries. These frameworks provide an abstraction of the underlying database access details, allowing developers to focus on implementing the business logic of the application. They have become increasingly popular with implementations in most modern programming languages such as Java, C#, Python, and Ruby [42, 151]. A report also shows that among the 2,164 surveyed Java developers, ORMs are the leading means of database access [30]. However, due to this abstraction, developers may face challenges when debugging database access problems. ORMs automatically generate SQL queries based on various ORM configurations (e.g., the relationship among object types) and the called ORM APIs. As a result, developers do not have direct control over how the SQL queries are generated by ORM. If there are issues with a generated SQL query, developers may have difficulties knowing how and where the problematic SQL query is generated in the application code [153, 31, 41], causing challenges in debugging database access problems.

Motivated by the above-mentioned importance and challenges of database access, in this thesis, we first conduct an empirical study of database access bugs in seven large-scale Java open-source applications that use relational database management systems. Specifically, by manually examining the bug reports and commit histories ranging from 5 to 16 years, we investigate and derive characteristics such as categories, root causes, impact, and occurrences of database access issues when

using two different popular database access technologies: manually constructing SQL queries and using ORM. Our empirical study provides motivations and guidelines for future research to help avoid, detect, and test database access bugs in database-backed applications.

To assist developers in debugging database access problems, we then propose an approach for locating the origin (i.e., the control flow path containing a sequence of method calls) that generates a given SQL query. Our hybrid approach leverages static analysis and information retrieval (IR) techniques. It achieves good localization accuracy and has a better localization result than the traditional text-based search baseline. Specifically, our approach achieves a Top@5 accuracy ranging from 78.3% to 95.5% for SQL queries in sessions, marking a 225% improvement over the baseline. For individual query logs, the Top@5 accuracy ranges from 59.1% to 100%, marking a 333% improvement compared to the baseline. The illustration also demonstrates the effectiveness of our approach in locating data access issues that generate problematic SQL queries (i.e., slow SQL queries and database deadlocks). Our results show the potential of using a combination of IR techniques and static analysis to help locate database-related issues.

1.2 Research Objective

Thesis Statement: Accessing the database and debugging the database access issues associated with problematic SQL queries can be challenging yet crucial tasks in database-backed applications when using different technologies. By analyzing the characteristics of these issues from large-scale applications and providing a localization tool, we can provide support to developers on understanding and locating database access issues.

Database access is central to database-backed applications and is crucial for their maintenance and quality. However, developers may encounter various challenges when using different technologies to access the database. For instance, developers need to carefully construct complex SQL queries when executing SQL queries or call ORM APIs when using ORM frameworks. Errors in SQL queries or misuse of ORM APIs can result in issues such as unexpected query results or runtime exceptions. Additionally, because ORM frameworks abstract the underlying SQL query generation and execution, developers may struggle to debug database access problems. If there is a database access issue in a generated SQL query, developers may have difficulties knowing how and where the SQL query is generated in the application code, causing challenges in debugging database access problems.

The goal of this thesis is therefore to help with the quality of database access. The thesis first aims to understand the maintenance issues of database access in database-backed applications. The

thesis focuses on the root cause, impact, and occurrence of database access issues when using two different and popular database access technologies: manually constructing SQL queries and ORM. Then, the thesis aims to aid database access issue diagnosis by locating where a problematic SQL query is generated in the source code.

1.3 Thesis Overview

This section presents an overview of the thesis, including a brief summary of each chapter.

1.3.1 Chapter 2: Background and Literature Review

In this chapter, we first present the background related to this thesis. We then discuss related work, including database access issues and adequacy of tests in database-backed applications.

1.3.2 Chapter 3: Studying Characteristics of Database Access Bugs in Java Applications

Database accesses are crucial for the maintenance and quality of database-backed applications. However, the inherent difference between database-backed applications and the underlying DBMS may lead to different bugs and maintenance challenges. Many prior works study the maintenance issues of database-backed applications from the perspective of syntactic or semantic errors in SQL queries, SQL anti-patterns, SQL code smells, and performance issues. Despite these efforts, there is limited research on understanding database access bugs using SQL queries or ORM frameworks in database-backed applications. In this chapter, we address this gap by conducting an empirical study on the characteristics of database access bugs in database-backed Java applications. We investigate 423 database access bugs collected from the issue tracking system of seven large-scale Java open source applications that use relational database management systems (e.g., MySQL or PostgreSQL). We generalize five categories (SQL queries, Schema, API, Configuration, SQL query result) of the root causes of database access bugs, containing 25 unique root causes. For each category, we thoroughly discuss the root cause, impact (e.g., crash), and how these bugs occur by manually examining the bug reports and commit histories. In addition, we find that the bugs pertaining to SQL queries, Schema, and API cover 84.2% of database access bugs across all studied applications. In particular, SQL queries bug (54%) and API bug (38.7%) are the most frequent issues when using JDBC and Hibernate, respectively. We believe that the categories of the root causes of database access bugs can be useful for developers to help them avoid pitfalls and serve as a checklist to help testers improve test scenarios that address specific database access bugs. Also, the distribution

of database access bugs between using JDBC and ORM frameworks provides complementary to developers in selecting database access technologies, which often require trade-offs.

1.3.3 Chapter 4: Localizing the Origin of SQL Queries in Database-Backed Web Applications

In database-backed web applications, developers often leverage Object-Relational Mapping (ORM) frameworks for database access. Rather than manually constructing SQL queries and embedding them in application code, ORM frameworks provide an abstraction of the underlying database access details, allowing developers to manipulate persistent data as if it were in-memory objects through APIs exposed by the ORM. However, due to the abstraction, developers may not know where and how a problematic SQL query is generated in the application code, causing challenges in debugging database access problems. In this chapter, we propose an approach to that addresses this challenge by locating where a SQL query is generated in the application code. Specifically, the approach takes the SQL queries in database logs and source code as the input, and outputs the most likely control flow paths for those SQL queries. The approach is a hybrid, using both static analysis and information retrieval (IR) techniques. It uses static analysis to infer the database access for every possible path in the control flow graph. Then, given a SQL query, it applies IR techniques to find the control flow path (i.e., a sequence of methods called in an interprocedural control flow graph) whose inferred database access has the highest similarity ranking. Evaluation on seven open source Java applications shows that our approach outperforms the baseline approach and achieves good localization results for both SQL queries in sessions and individual SQL queries. We also conduct a study to illustrate how our approach can be used for locating issues in the database access code.

1.3.4 Chapter 5: Thesis Contributions and Future Work

In this chapter, we summarize the contributions of this thesis and discuss several potential directions for future work.

1.4 Thesis Contributions

The contributions of this thesis are as follows:

- We conduct an empirical study of database access bugs spanning 5 to 16 years in seven large-scale Java open-source applications that use relational database management systems. We investigate and derive characteristics such as categories, root causes, impact, and occurrences of database access issues when using popular database access technologies. Our findings provide

motivations and guidelines for future research to help avoid, detect, and test database access bugs in database-backed applications (Chapter 3).

- To the best of our knowledge, we conducted the first empirical study of database access bugs across JDBC and Hibernate. The distribution of database access bugs between using JDBC and ORM frameworks provides complementary to developers in selecting database access technologies, which often require trade-offs (Chapter 3).

- To assist developers in debugging database access problems, we propose an approach that leverages both static analysis and information retrieval (IR) techniques to locate where a SQL query is generated in the application code. Our approach outperforms the baseline approach and achieves good localization results for different levels of granularity in SQL queries (Chapter 4).

1.5 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses the background and literature review of this thesis. Chapter 3 presents our study on characteristics of database access bugs in Java applications. Chapter 4 describes our approach to localizing the origin of SQL queries in database-backed web applications. Chapter 5 concludes the thesis and discusses potential directions for future work.

Chapter 2

Background and Literature Review

In this chapter, we first present the background information related to our research. We then present the related works of our research on database-backed applications, including database access quality issues associated with using SQL queries or ORM frameworks. We also present the related works on adequacy of tests in database-backed applications.

2.1 Background

Background of Database Access. Database access plays a central role in database-backed applications. Important business logic in such applications requires selecting, inserting, and updating data in database management systems (DBMSs), such as MySQL [117]. While objects in database-backed applications are often implemented in object-oriented programming languages such as Java, database records are rows in tables defined by the database schema. The mapping between application objects and database records can be complex and usually involves some impedance mismatch (i.e., conceptual differences between object-oriented programming and relational databases) [37]. Due to this mismatch, database records need to be converted into corresponding objects in the application.

Developers often rely on two main technologies to access the underlying database and convert database records into objects in applications. The *first* technology is constructing SQL queries manually and executing the SQL query by calling standard database connectivity interfaces [47] which provides an abstraction for different DBMSs. In Java, the standard database connectivity interfaces are implemented as Java Database Connectivity (JDBC) APIs. Other programming languages also offer similar database access APIs for executing SQL queries, such as DB-API for Python [2] and mysql2 for Ruby [3]. Developers call JDBC APIs to send the manually constructed SQL queries to

DBMSs. Subsequently, the applications retrieve the query results, and developers manually write code to convert them into objects. The *second* technology is Object-Relational Mapping (ORM) frameworks, which provide developers a conceptual abstraction for mapping database records to objects in object-oriented languages [38]. When using Hibernate (the most popular ORM framework in Java [30]), developers only need to configure the mapping between entity classes (i.e., the class mapped to a database table) and database tables, and then call Hibernate APIs. Such a mapping (configuration) allows Hibernate to automatically generate SQL queries executed by DBMSs and convert the object to/from the database record. Other programming languages also offer similar abstractions within their ORM frameworks, such as Django for Python [1] and Ruby on Rails for Ruby [4]. Compared to JDBC, ORM allows developers to focus on developing the business logic without worrying too much about the database access details. When using JDBC and Hibernate, developers can choose whichever underlying DBMS they want to use. These two database access frameworks support most, if not all, relational database management systems. Developers can even switch between the underlying DBMS if needed.

To better understand these two database access technologies, we take Java as an example and compare the database access code using JDBC and Hibernate. Figure 1a shows an example of database access using JDBC, where developers manually construct the SQL query and call the JDBC API `statement.executeQuery(query)` to issue the SQL query to the DBMS (Lines 2-3). After the SQL query is executed by the DBMS, developers retrieve the database record in query result `resultSet` and convert it into an `address` object (Lines 6-12). Figure 1b shows an example of database access using Hibernate, which achieves the same functionality as the code in Figure 1a. The entity class `Address` (annotated with `@Entity`) is mapped to the table `address` in the DBMS using the annotation `@Table`. Developers need to set up the configuration to map each field in the `Address` entity to the database column (annotated with `@Column`) (Lines 5-10). When calling the Hibernate API `session.get(Address.class, 1)` (Line 14, which gets the address in the DMBS with an ID of 1), Hibernate automatically generates the SQL query, which is then sent to and executed by the DBMS. Hibernate then automatically serializes the query results to an `address` object. Then, developers can call `address.setStreet(“First Street”)` followed by `session.update(address)` to update the corresponding database record.

Despite the wide usage of database-backed applications, their development may be challenging due to the co-evolution of database schema and application code [125]. For instance, developers may be unaware of the table column name change in databases to update corresponding code in applications. Thus, accessing the database may cause runtime exceptions when the column specified in the SQL query does not exist in the database. Even worse, developers may face different challenges to access the database when using different technologies. When using JDBC, developers

```java
// Call JDBC API to execute the SQL query
String query = "SELECT * FROM address WHERE ADDR_ID = 1"
ResultSet resultSet = statement.executeQuery(query);

// Retrieve the database record in query result ResultSet
while (resultSet.next()) {
   // Convert the database record to Java object address
   Long id = resultSet.getLong("ADDR_ID");
   String street = resultSet.getString("ADDR_STREET");
   ...
   Address address = new Address(id, street, ... );
}
```

(a) JDBC

```java
// Configure the mapping
@Entity
@Table(name = "address")
public class Address {
   @Column(name = "ADDR_ID")
   private long id;

   @Column(name = "ADDR_STREET", length = 40 )
   private String street;
   ...
}

// Call Hibernate API to fetch the database record
Address address = (Address) session.get(Address.class, 1);
```

(b) Hibernate

Figure 1: Examples of database access using JDBC and Hibernate.

need to carefully construct complex SQL queries which should be free from syntax errors and be executed successfully under the database schema constraint. On the other hand, developers may unintentionally misuse the ORM APIs because ORM frameworks abstract the underlying SQL query generation and execution. For instance, when calling ORM APIs, the generated SQL queries may retrieve unused/unnecessary data from the database, thereby causing performance bugs [38].

2.2 Literature Review

2.2.1 Paper Selection

Database access quality involves many research areas. In this thesis, we mainly review papers related to three aspects: 1) database access quality issues when using SQL queries; 2) database access quality issues when using ORM frameworks; and 3) research on testing database-backed applications. We select research papers from renowned venues in the field of software engineering and databases (as shown in Table 1). Based on the results of our search, we identified a total of 41 relevant studies, including 38 research papers and 3 books, covering 20 years of research (2004 $\sim$ 2023).

Table 1: Name of the conferences and journals as venues for the literature review.

Field	Conference/Journal Name	Abbreviation
SE	International Conference on Software Engineering	ICSE
SE	International Conference on the Foundations of Software Engineering	FSE
SE	International Conference on Automated Software Engineering	ASE
SE	International Symposium on Software Testing and Analysis	ISSTA
SE	IEEE Transactions on Software Engineering	TSE
SE	ACM Transactions on Software Engineering and Methodology	TOSEM
SE	International Conference on Software Maintenance and Evolution	ICSME
SE	International Conference on Quality Software	QSIC
SE	Software Testing, Verification and Reliability	STVR
SE	International Conference on Program Comprehension	ICPC
SE	International Conference on Software Testing, Verification and Validation	ICST
SE	Asian Conference on Programming Languages and Systems	APLAS
SE	Information and Software Technology	IST
SE	International Conference on Source Code Analysis	SCAM
SE	International Conference on Mining Software Repositories	MSR
SE	Conference on Innovation and Technology in Computer Science Education	ITiCSE
SE	Engineering Reports	Engineering Reports
SE	Science of Computer Programming	Sci. Comput. Program.
SE	International Workshop on Mutation analysis	Mutation
DB	International Conference on Management of Data	SIGMOD
DB	International Conference on Data Engineering	ICDE
DB	ACM on Conference on Information and Knowledge Management	CIKM

2.2.2 Database access quality issues when using SQL queries

When accessing the database in database-backed applications, developers need to carefully construct SQL queries that are free from syntax errors and correctly embed them in the host language (e.g., Java). However, even syntactically correct SQL queries may still encounter issues, resulting in database access issues. The related work on database access issues can be summarized into three aspects: 1) syntactic or semantic errors in SQL queries, 2) SQL anti-patterns, and 3) SQL code smells. Below, we summarize the related work for each aspect.

Syntactic or semantic errors in SQL queries. Gould et al. [61] proposed a static analysis approach for syntax analysis and type checking within SQL queries to verify the correctness of SQL query strings in database-backed applications. Unlike syntactic errors, a semantic error in an SQL query means that it is syntactically correct but does not return the intended results when executed. For example, a SQL query such as SELECT * FROM EMP WHERE JOB = 'CLERK' and JOB = 'MANAGER' contains an inconsistent condition, resulting in an empty result set. However, no task would query the database to produce an always empty result. Brass and Goldberg [28] identified 37 types of semantic errors in SQL queries and developed a tool called SQLLint [5] to detect them. Ahadi et al. [17] also investigated semantic errors in seven types of SQL SELECT statement queries and the reasons behind them. They found that semantic errors are much harder to correct compared to syntactic errors in SQL queries. Annamaa et al. [20] proposed a SQL syntax analyzer to statically analyze SQL queries embedded in Java programs to detect syntactic errors in SQL queries. They also proposed a testing facility to generate sample SQL statements from embedded SQL queries and perform semantic validation of them on a running database engine.

SQL anti-patterns. Anti-patterns are design decisions intended to solve a problem but often lead to other issues by violating fundamental design principles. In the context of SQL, anti-patterns arise from violating practices that suggest the best way to retrieve and manipulate data using SQL, which may affect the performance, maintainability, and accuracy of database-backed applications [52]. One example of an SQL anti-pattern is using the INSERT statement without explicitly listing column names. For instance, a SQL query like INSERT INTO Bugs VALUES (DEFAULT, CURDATE(), 'New bug', 'Test T987 fails...') relies on implicit column names and gives values for all columns in the same order that columns are defined in the table. However, if a new column is added to the Bugs table, the SQL query may produce an error because the value count would not match the column count anymore. Karwin [77] provided an overview of SQL design anti-patterns, while Alshemaimri et al. [18] categorized SQL anti-patterns in their survey. Arzamasova et al. [24] analyzed anti-patterns in SQL query logs and proposed a framework to discover and resolve these anti-patterns. Dintyala et al. [52] presented a holistic toolchain called SQLCheck for automatically finding and fixing anti-patterns

in database applications. Shao et al. [128] conducted a literature survey and reported 34 database access performance anti-patterns in total. Lyu et al. [86] proposed a static analysis approach, SAND, to detect SQL anti-patterns in mobile apps.

SQL code smells. Code smells indicate software design problems that harm software quality. Similarly, SQL code smells are issues within SQL code resulting from query misuses, such as excessively long or short identifiers [126]. Redgate [126] documented 119 SQL code smells. Nagy and Cleve [102] mined Stack Overflow questions to identify error-prone patterns in SQL queries. They also proposed a static analysis approach to detect SQL code smells in queries extracted from Java code [103, 104]. Gonçalves de Almeida Filho et al. [60] investigated the prevalence and co-occurrence of SQL code smells in PL/SQL projects. An empirical study by Muse et al. [100] investigated the prevalence and evolution of SQL code smells in open-source, data-intensive systems. The study finds that SQL code smells are prevalent and persist in these systems, independent of traditional code smells. In addition to SQL code smells, Sharma et al. [129] presented a catalog of 13 database schema smells (i.e., smells that arise due to poor schema design). They also developed a tool called DbDeo to detect database schema smells and found that 'index abuse' is the most frequent one.

Most prior research focuses on the quality of database access by statically analyzing the quality of SQL queries. Despite these efforts, there is still limited research on understanding database access bugs related to using SQL queries or ORM frameworks. Database access bugs in database-backed applications during runtime may be different from the syntactic or semantic errors in isolated SQL queries since database access leverages SQL queries embedded within the application code or generated by the ORM frameworks to interact with DBMSs. On the other hand, SQL anti-patterns or SQL code smells are potential indications of quality issues (not necessarily bugs), and allow programs to execute correctly with quality problems such as poor performance. In contrast, database access bugs may lead to unexpected program behaviors, with diverse causes and impacts. For instance, as database-backed applications and their underlying database evolve (e.g., modifying a table column name), the specified column in the SQL query might no longer exist, resulting in severe problems like exceptions when executing the SQL query. This thesis addresses this gap by conducting an empirical study on the characteristics (e.g., bug occurrence and root cause) of database access bugs in database-backed applications (in Chapter 3).

2.2.3 Database access quality issues when using ORM frameworks

Since ORM frameworks abstract the underlying SQL query generation and execution, developers may unintentionally misuse the ORM APIs to generate problematic SQL queries, resulting in database access issues. One example of such an issue is retrieving unused or unnecessary data from

the database [38]. Many prior studies have focused on detecting and analyzing database access quality issues in applications that use ORM frameworks. Chen et al. [38, 40] proposed an automated framework to detect and prioritize both performance and functional ORM anti-patterns. They also identified four types of redundant data access and found that eliminating them can improve SQL execution time by up to 92% [41]. Yan et al. [150] studied database-related performance inefficiencies in web applications built using the Ruby on Rails ORM framework. Yang et al. [151] identified several performance anti-patterns and proposed detection algorithms based on static analysis. They also developed a tool called PowerStation to automatically detect and fix ORM-related performance issues in database-backed web applications [152]. Chen et al. [36] identified and cataloged 17 performance anti-patterns for ORM applications written in PHP. They find that the response time of the applications significantly reduces after refactoring these anti-pattern instances. Huang et al. [71] proposed a static analysis-based tool called HBSniff to detect 14 code smells in Java source code using Hibernate ORM framework.

Most prior studies focus on detecting database access issues at the code level (i.e., anti-patterns). Anti-patterns in ORM code may lead to generating inefficient or incorrect SQL queries. However, one limitation of the prior approaches is that they focus on detecting issues based on predefined/known anti-patterns [128] in database-backed applications and cannot detect issues that do not belong to any of the predefined anti-patterns. In this thesis, given a potentially problematic SQL query, we locate the code path that generates the given query (in Chapter 4). Hence, we complement prior approaches by identifying the code that may result in generating problematic SQL queries.

2.2.4 Adequacy of tests in database-backed applications

Prior works have proposed different coverage criteria to measure the adequacy of tests for database-backed applications, e.g., SQL commands [66], SQL queries and clauses [135, 140], and database schema constraints [90]. Halfond and Orso [66] introduced a database interaction testing adequacy criteria based on *command-form* coverage which takes into account variants of SQL commands. Suárez-Cabal and Tuya [135] introduced a coverage metric for `SELECT` queries while Tuya et al. [140] proposed a form of predicate coverage criterion for SQL queries by considering the coverage of several clauses like `JOIN`, `WHERE`, `HAVING`, `GROUP BY`, etc. Mcminn et al. [90] proposed a family of coverage criteria for testing the integrity constraints in a relational database schema. However, these coverage criteria are defined separately for embedded SQL queries.

Other works have applied mutation testing to SQL queries to assess the adequacy of tests in database-backed applications and proposed mutation operators for SQL queries [75, 127]. Chan et al. [34] proposed seven SQL mutation operators based on the enhanced entity-relationship model. Tuya et al. [138, 139] proposed a set of mutation operators for SQL queries and integrated these operators

into a tool called SQLMutation that automatically generates mutants for SQL queries. Zhou and Frankl [157] extended the work by Tuya et al. and applied mutation testing to database application programs by performing those mutation operators on the SQL queries in Java/JDBC applications. Gupta et al. [63] proposed a set of join/outer-join mutations that model common programmer errors. Kapfhammer et al. [76] and Wright et al. [148] also proposed operators for introducing SQL query faults that violate database schema constraints. However, all the above mutation operators for SQL queries are proposed aiming at covering SQL features or SQL syntax and semantics without considering real SQL query bugs. Our study of database access bugs (especially the SQL queries and database schema-related bugs) in large-scale open-source applications (in Chapter 3) can be used as a complementary aid for designing mutation operators and SQL mutants [91]. Furthermore, our findings show that several database access bugs are introduced when calling database access APIs (21.7%) or converting SQL query results (6.8%), which calls for the study of mutation for host languages (e.g., Java) that access databases by issuing SQL queries.

2.3 Chapter Summary

In this chapter, we present a literature review on the state-of-the-art research regarding database access quality in database-backed applications. The related research papers mainly focus on the database access quality issues from the perspective of syntactic or semantic errors in SQL queries, SQL anti-patterns, and SQL code smells. However, there is still a lack of studies toward understanding database access bugs that occur when using SQL queries or ORM frameworks in database-backed applications. These database access bugs may differ from those quality issues and can lead to unexpected program behavior (e.g., crashes) with diverse causes and impacts. On the other hand, researchers often leverage static analysis to detect database access issues at the code level (e.g., anti-patterns). However, this approach cannot detect issues that do not belong to any of the predefined anti-patterns. Motivated by the findings of our literature review, we first conducted an empirical study on the characteristics (e.g., bug occurrence and root cause) of database access bugs in database-backed applications (in Chapter 3). We then propose an approach to help developers locate database access issues by identifying the code path that generates potentially problematic SQL queries (in Chapter 4).

Part II

Understanding and Locating Database Access Code Quality Issues

Chapter 3

Studying the Characteristics of Database Access Bugs in Java Applications

Database-backed applications rely on the database access code to interact with the underlying database management systems (DBMSs). Developers may face different challenges in accessing the database when using different technologies (e.g., executing SQL queries or calling ORM APIs). Although many prior studies focus on database access issues such as SQL anti-patterns or SQL code smells, limited research addresses database access bugs during the maintenance of database-backed applications. In this chapter, we empirically investigate 423 database access bugs collected from seven large-scale open source Java applications that use relational database management systems (e.g., MySQL or PostgreSQL). We study the occurrence of the bugs and find that the number of reported database and non-database access bugs shares a similar trend, but their modified files in bug fixing commits are different. In addition, we generalize categories of the root causes of database access bugs, including five main categories (SQL queries, Schema, API, Configuration, SQL query result) and 25 unique root causes. For each category, we thoroughly discuss the root cause, impact (e.g., crash), and how these bugs occur by manually examining the bug reports and commit histories. We find that the bugs pertaining to SQL queries, Schema, and API cover 84.2% of database access bugs across all studied applications. In particular, SQL queries bug (54%) and API bug (38.7%) are the most frequent issues when using JDBC and Hibernate, respectively. Finally, we discuss the implications of our findings for developers and researchers.

An earlier version of this chapter is published at ACM Transactions on Software

Engineering and Methodology, 33(7), September 2024. doi: 10.1145/3672449. [84]

3.1 Introduction

From online shopping to social media, many applications need to store and access data at their back-end for rich functionalities and better user experiences. Such database-backed applications are built around the database access to interact with database management systems (DBMSs), such as MySQL, to store and retrieve data values. These database accesses are crucial for the maintenance and quality of database-backed applications.

Developers often build database-backed applications with object-oriented programming languages such as Java, Python, C#, PHP, and C++ [101, 122]. Since object-oriented programming is a different paradigm compared to relational databases, developers use different technologies to ease database access by abstracting persistent data as objects. Developers often rely on two main access technologies: (i) execution of a Structured Query Language (SQL) query (e.g., using JDBC) and manually converting the results to objects; and (ii) using Object-Relational Mapping (ORM) frameworks, which automatically generate SQL queries and convert the results to objects based on various object-database mapping configurations. However, the inherent difference between database-backed applications and the underlying DBMS may lead to different bugs and maintenance challenges. For instance, since the syntax of SQL queries is not checked during compile time, syntax errors in SQL queries may lead to production issues. On the other hand, developers may unintentionally misuse the ORM APIs because the ORM framework hides both the generation of the underlying SQL query and its execution. As an example, when issuing calls to ORM APIs, the generated SQL queries may retrieve unused/unnecessary data from the database, thereby causing performance bugs [38].

There are many prior works that study the maintenance issues of database-backed applications from the perspective of syntactic or semantic errors in SQL queries [28, 17], SQL anti-patterns [77, 24, 52, 18], SQL code smells [126, 129, 100], and performance issues [38, 150, 151, 128]. Specifically, Brass and Goldberg [28] proposed a list of semantic errors in SQL queries. A recent survey by Alshemaimri et al. [18] summarized categories of SQL anti-patterns and framework-specific (e.g., ORM) anti-patterns. Redgate [126] documented 119 SQL code smells while Shao et al. [128] conducted a literature survey and reported 34 database access performance anti-patterns in total. Despite these efforts, there is limited research on understanding database access bugs using SQL queries or ORM frameworks. Database access bugs in database-backed applications during runtime may be different from the syntactic or semantic errors in separate SQL queries since database access leverages SQL queries embedded within the application code or generated by the ORM frameworks to interact with DBMSs. Unlike SQL anti-patterns or SQL code smells, which allow programs to

execute correctly but have quality problems such as poor performance or indicate the presence of quality problems but not necessarily bugs, respectively, database access bugs may cause severe problems like crashes. Our work addresses this gap by providing categories of database access bugs from the issue tracking system and highlighting their root causes. Inspired by previous bug characterization studies [72, 128], we define the root cause as a human mistake in the program code, database schema, or configuration that causes database access errors.

In this study, we conduct an empirical study to understand the characteristics and causes of database access bugs in Java database-backed applications, since Java is one of the most popular programming languages [59, 15] used by millions of developers worldwide [118, 132]. We focus on studying the systems that use relational database management systems (e.g., MySQL or PostgreSQL) due to their wide adoption and frequent use in handling complex data requests[1]. We consider all types of database access bugs (e.g., not limited to performance issues that were the main focus in prior studies [150, 151, 128]) and consider the bugs that occur in applications that use two different types of technologies (i.e., JDBC and ORM). We conducted an empirical study on seven popular and large-scale open-source Java database-backed applications. These applications use either the Java Database Connectivity (JDBC) or the Hibernate ORM framework for database access. JDBC is part of the official Java Development Kit (JDK) for accessing the DBMS and Hibernate is one of the most popular Java ORM frameworks [30].

We collected a statistically significant sample of 5,323 fixed bug issues from the issue tracking systems of studied applications, of which 423 were manually identified as database access bugs and 4,900 are non-database access bugs. We performed a quantitative study to compare the database access bugs to non-database access bugs to study their characteristics (i.e., reported trends). Then, we performed a qualitative study on 423 database access bugs to examine their root causes. We manually examined the bug reports and commit histories of the database access bugs to understand how developers discuss/fix these bugs. The goal of the manual analysis is to identify the root cause behind the bugs and the output is categories of the root causes of database access bugs related to JDBC or Hibernate.

In particular, we seek to answer the three following research questions (RQs):

RQ1 (*Bug occurrence*): What is the trend in the number of reported database access bugs? We find that database access bugs are reported throughout the life cycle of the applications. These observations indicate that database access bugs are common and the maintenance of database access code requires continuous attention. We also find that developers modify different sets of files in bug fixing commits for database access bugs compared to non-database access bugs, which indicates that database access bugs may have their own unique characteristics and motivates further research

[1] https://www.ibm.com/cloud/blog/sql-vs-nosql

to examine their root causes.

RQ2 (*Root cause*): What are the root causes of database access bugs? We generalize categories of the root causes of database access bugs. We derived the category by manually studying 423 database access bugs and identified 25 unique root causes. We find that most of the database access bugs cause problems like runtime exceptions or the return of unexpected query results to users. While a few of these bugs have been identified in the prior work [28, 52, 40], the majority of them have not yet been examined in the past.

RQ3 (*Bug category*): How do categories of database access bugs prevail with different database access technologies? To determine whether certain categories of database access bugs arise frequently and whether this is dependent on database access technologies, we compared the percentage of categories of bugs across JDBC and Hibernate. Our study reveals that *SQL queries*, *schema*, and *API* bugs cover 84.2% database access bugs across all studied applications. Moreover, *SQL queries* bug (54%) and *API* bug (38.7%) are the most frequent issues when using JDBC and Hibernate, respectively.

The main contributions of this study are as follows:

- To the best of our knowledge, we conduct the first empirical study of database access bugs in database-backed applications.
- We find that the number of reported database and non-database access bugs share a similar trend throughout the life cycle of database-backed applications. However, their modified files in bug fixing commits are different. This implies that they are not necessarily co-located and database access bugs may have different causes and fixes as compared to non-database access bugs.
- We generalize categories of the root causes of database access bugs into five main categories, containing 25 unique root causes, by thoroughly studying 423 database access bugs. We find that *SQL queries*, *schema*, *API* bugs cover 84.2% database access bugs across all studied applications. We also find that *SQL queries* bug (54%) and *API* bug (38.7%) are the most frequent issues when using JDBC and Hibernate, respectively. We provide a discussion and implication of our findings for future work.

Overall, we conduct an empirical study of database access bugs in database-backed Java applications that use relational database management systems. In particular, we study the characteristics of database access bugs and the categories of their root causes. Our empirical study provides motivations and guidelines for future research to help avoid, detect, and test database access bugs in database-backed applications. Our dataset is publicly available [10].

Chapter Organization. The rest of this chapter is organized as follows. Section 3.2 describes the studied applications and our data collection approach. Section 3.3 presents our detailed results and Section 3.4 further discusses them and provides actionable implications. Next, we discuss possible threats to validity (Section 3.5) and survey related work (Section 3.6). Finally, Section 3.7 concludes the chapter.

3.2 Empirical Study Setup

In this section, we present the setup for our empirical study. In particular, we describe our process of collecting studied applications and database access bugs.

3.2.1 Collecting Studied Applications

We focus our study on database access bugs from open-source applications implemented with Java, which is one of the most popular programming languages [59] used by millions of developers worldwide [118, 132]. Besides, applications in Java have been studied by many prior studies [44, 119, 42, 39, 105, 21].

We apply three selection criteria to select the studied applications. *First*, we pick the top 100 most popular Java applications that use database technology from GitHub based on the number of stars. We filter the applications using database technology by manually examining the application's description and wiki on GitHub. *Second*, a candidate application should use an issue tracking system (e.g., Jira) and contain database access bug reports so that we can study real-world bugs that occur when accessing the database. *Finally*, the application should be actively maintained, having a long revision history (having more than 1,000 commits) and at least 100 fixed bug reports.

We end up with seven open-source applications in total that satisfy our selection criteria. Table 2 shows an overview of the studied applications, such as the number of stars, lines of code (LOC), and commits. The studied applications are from various domains and four use JDBC to access the database while the other three use Hibernate. `BroadleafCommerce` [29] is an enterprise e-commerce framework while `metasfresh` [92] is an enterprise resource planning (ERP) system. `Openfire` [107] is a real-time collaboration (RTC) system that supports instant messaging using the open communication protocol. `ADempiere` [16] is an enterprise business suite integrated with ERP, customer relationship management, and supply chain management. `DBeaver` [51] is a universal and multi-platform database tool which supports various popular DBMSs. `dotCMS` [53] is a content management system (CMS) while `OpenMRS` [108] is a widely used patient-based electronic medical record (EMR) system. All of the studied applications have been developed over a period of 5 to 16 years.

Table 2: The studied applications and bug issues.

Application	Persistence	Description	Stars(K)	LOC(K)	Commits	Study Period	# Fixed bug issues	# Studied bug issues	# Database access bugs
`BroadleafCommerce`	Hibernate	E-commerce	1.5	197	17,381	03/13 ∼ 03/17	593	593	57/593 (9.6%)
`metasfresh`	JDBC	ERP System	0.9	1,610	52,927	06/16 ∼ 03/21	712	712	27/712 (3.8%)
`Openfire`	JDBC	RTC Server	2.4	117	10,034	08/05 ∼ 04/21	749	749	41/749 (5.5%)
`ADempiere`	JDBC	Business Suite	0.6	879	16,006	09/15 ∼ 03/21	866	866	63/866 (7.3%)
`DBeaver`	JDBC	Database Tool	22.5	430	21,176	10/15 ∼ 06/21	3,203	801	106/801 (13.2%)
`dotCMS`	Hibernate	CMS	0.6	522	18,317	03/12 ∼ 05/21	4,607	867	50/867 (5.8%)
`OpenMRS`	Hibernate	EMR System	1.0	128	11,530	03/06 ∼ 03/21	2,359	735	79/735 (10.7%)

*Note that we study the issues of BroadleafCommerce until 03/2017 because developers do not actively maintain the issue tracking system for the open source version. However, developers are still actively developing the application.

3.2.2 Collecting Database Access Bugs

We collect the database access bugs from the issue tracking system of the studied applications. We first collect fixed bug issues and filter the issues using database-related keywords. Finally, we manually verify whether the identified issues are related to database access bugs. Below, we discuss our bug collection process in detail.

Collecting studied bug issues. We collect the issues with the type of *bug* and fix status of *fixed* and *resolved* in each application during the study period, from the time they were first reported to 06/2021. Note that, we study the issues of `BroadleafCommerce` until 03/2017 because developers do not actively maintain the issue tracking system for the open source version of `BroadleafCommerce`. However, developers are still actively developing the application [29]. Table 2 shows the number of fixed bug issues for each application. From these, we identified the studied bug issues. For applications with less than 1,000 fixed bug issues (i.e., the first four applications), we analyze all fixed bug issues. However, for other applications which have 2,359 to 4,607 fixed bug issues (i.e., DBeaver, dotCMS, and OpenMRS), it is not feasible to manually study all bug reports. To address this, we randomly select a statistically significant sample from the fixed bug issues with a 95% confidence level and a 3% margin of error. This sampling approach results in 801, 867, and 735 studied bug issues for DBeaver, dotCMS, and OpenMRS, respectively.

Filtering & verifying studied bug issues. We filter the studied issues by searching for database-related keywords in each issue's title, description, and comments to get the database-related issues. The database-related keywords are concluded by manually examining 50 random database-related issues from the issue tracking system and are listed below along with matching text (underlined) examples:

- database: "the entry is added in the <u>database</u>"

- constraint: “a foreign key constraint fails”
- MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server: “PubSubManager: DELETE FROM ofPubsubItem LEFT JOIN breaks MySQL”
- SQL: “You have an error in your SQL syntax”
- Hibernate: “Hibernate startup errors”
- JDBC: “I have confirmed that this is an issue with the mysql JDBC driver version.”

Next, we verify the database-related issues manually to examine the database access bug or non-database access bug because our heuristic approach (i.e., filtering by keywords) may result in false positives. For instance, BroadleafCommerce#378 reports an issue that contains the keyword “database” in the sentence “the data is saved appropriately in the database, but the list grid value for the column is not immediately updated”. However, according to the sentence, the issue is not a database access bug since the program accesses the database correctly. Instead, the issue is caused by calling an API incorrectly which leads to incorrect UI (i.e., list grid) refresh.

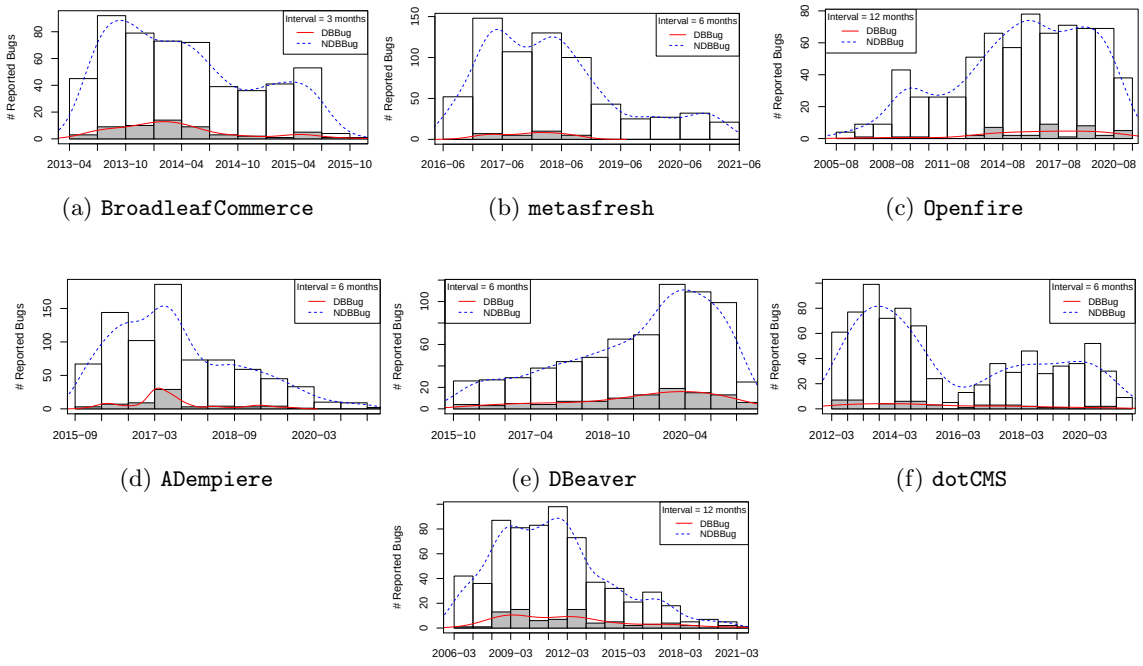
After the filtering and verifying process, we end up with manually verified database access bugs for each studied application, shown in the last column of Table 2. Other bugs in the studied bug issues are identified as non-database access bugs. The percentages of database access bugs among studied bug issues range from 3.8% to 13.2%. In total, we collected 423 database access bugs, among which 186 are related to Hibernate and 237 are related to JDBC. We also collected 4,900 non-database access bugs.

3.3 Empirical Study Results

In this section, we present the results of our empirical study by answering the research questions (RQs).

3.3.1 RQ1: What is the Trend in the Number of Reported Database Access Bugs?

Motivation. In this research question, we study when database access bugs are reported in the studied applications. In particular, we want to examine whether database access bugs are more likely to occur during a specific time period. For example, are most database access bugs reported at the beginning of the development history, with fewer reported as the application evolves, or are the bugs reported throughout the development history? We are also interested in the trend of the number of database access bugs reported, because database access is critical to database-backed



(a) `BroadleafCommerce` (b) `metasfresh` (c) `Openfire`

(d) `ADempiere` (e) `DBeaver` (f) `dotCMS`

(g) `OpenMRS`

Figure 2: The trend of reported database access bugs (DBBug) and non-database access bugs (NDBBug) across the studied applications. The reported bugs are aggregated at a fixed time interval according to their reporting time in each application.

applications and any failures related to database access may have disastrous results [87]. Knowing when database access bugs are reported is the first step toward better debugging and maintenance of database-backed applications.

Approach. We study how many and when the database access bugs collected in Section 3.2 are reported across the study period. For each studied application, we study the number of reported bugs. For each bug, we take the creation time of the bug report as the reporting time. To gain a more comprehensive understanding of the quantitative magnitude of database access bugs, we further study the correlation between the reported numbers of database access bugs and non-database access bugs throughout the study period (i.e., every three or six months). We choose to use Spearman's rank correlation r_s because it is a non-parametric correlation test and does not have an assumption on the underlying data distribution [160]. We classify the strength of the relationship between the number of database access bugs and non-database access bugs to *zero* (0), *weak* ($\pm$ 0.1 to $\pm$0.3), *moderate* ($\pm$ 0.4 to $\pm$ 0.6), *strong* ($\pm$ 0.7 to $\pm$ 0.9), and *perfect* ($\pm$ 1) according to the r_s value threshold as per prior work [50].

Results. We find that the number of database and non-database access bugs share a similar trend.

Table 3: Spearman's rank correlation (r_s) between the number of reported database and non-database access bugs across the study period. The reported bugs are aggregated using two time intervals (i.e., 3 months and 6 months).

Application	r_s	
	Interval = 3 months	**Interval = 6 months**
`BroadleafCommerce`	0.9	0.9
`metasfresh`	0.9	0.9
`Openfire`	0.5	0.7
`ADempiere`	0.7	0.9
`DBeaver`	0.9	0.9
`dotCMS`	0.4	0.5
`OpenMRS`	0.5	0.5

Figure 2 shows the trend of reported bugs during the study period. The x-axis represents the time interval, and the y-axis represents the number of reported bugs. For better visualization, we aggregate the reported bugs at a fixed time interval according to their reporting time in each application (e.g., interval = 3 months in `BroadleafCommerce`) and count the number of reported bugs within every time interval. We see that database access bugs are reported throughout the study period. Although sometimes more database access bugs are reported, not all of them are reported during a specific time (e.g., when the application is first released). This finding indicates that database access code maintenance [125, 42, 94] is a continuous process for database-backed applications and requires continuous attention. We also use the density curves (i.e., the smooth lines) to fit the distribution of reported bugs. For each application, we find that both database access bugs and non-database access bugs have a similar trend - the density value increases at the beginning, reaches a peak, and decreases thereafter.

We further apply correlation analysis (i.e., Spearman's rank correlation) to verify if there is a consistent trend between the number of reported database and non-database access bugs. We calculate the Spearman's rank correlation coefficient r_s between the number of reported database and non-database access bugs based on two time intervals (i.e., 3 months and 6 months) and report the results of r_s in Table 3. We find that r_s values of four studied applications are between 0.7 to 0.9 (*strong*) while r_s values of two studied applications are between 0.4 to 0.6 (*moderate*). The results indicate that the number of reported database access and non-database access bugs have a *moderate* to *strong* correlation: increasing or decreasing at the same time.

Discussion. We find that the number of reported database and non-database access bugs share a similar trend. Hence, we further investigate whether the reported database and non-database access

Table 4: The number of unique modified files in bug fixing commits for reported database access bugs (DBBug) and non-database access bugs (NDBBug), the percentage (Pct.) of modified files for DBBug across modified files for all bugs (i.e., DBBug and NDBBug), and the number of common modified files (COM) shared between DBBug and NDBBug.

Type	Broadleaf-Commerce			metasfresh			Openfire			ADempiere			DBeaver			dotCMS			OpenMRS		
	#	Pct.	#COM	#	Pct.	#COM	#	Pct.	#COM	#	Pct.	#COM	#	Pct.	#COM	#	Pct.	#COM	#	Pct.	#COM
DBBug	119	16%	64	72	2%	36	33	7%	23	63	8%	31	182	20%	88	133	16%	33	149	18%	90
NDBBug	645			2946			452			772			735			704			664		

bugs occur at similar locations by examining their modified files in bug fixing commits. Table 4 shows the number of unique modified files in bug fixing commits between database access bugs and non-database access bugs. The database access bugs occur in a small range of source code files (the number of modified files is between 33 to 182), while the non-database access bugs occur in a large range of source code files (the number of modified files is between 452 to 2,946). In particular, the percentage of modified files for database access bugs across modified files for all bugs is between 2% to 20%. We also observe that there are 23 to 90 common modified files (COM) shared between database access bugs and non-database access bugs. The reason is that database access code may share common files with other code (e.g., business logic code). As found in a prior study by Qiu et al. [125], database schema co-evolves with the application code, which may explain the correlation between the number of reported bugs. However, our findings show that database access bugs may occur in different source code files compared to non-database access bugs. In other words, database access bugs require further research to understand their root causes and attention from the research community. In the next RQ, we manually study database access bugs to understand and create categories of the root causes.

Database access bugs are reported throughout the development history of the applications. While the number of reported database and non-database access bugs share a similar trend, their modified files in bug fixing commits are different. This implies that they are not necessarily co-located and database access bugs may have different causes and fixes as compared to non-database access bugs.

3.3.2 RQ2: What are the Root Causes of Database Access Bugs?

Motivation. In RQ1, we find that developers modify different files when fixing database and non-database access bugs. Thus, database access bugs may have their own unique characteristics of

root cause and impact. In this RQ, we manually study database access bugs to uncover their root causes. Our goal is to create categories of the root causes that may inspire future research and help practitioners avoid common pitfalls. *For researchers*, the category could guide in designing research tools to improve the quality of database-backed applications. *For practitioners*, the category could serve as a checklist to define test scenarios that address specific bug types in database access.

Approach. For the 423 database access bugs collected in Section 3.2, we manually study their bug reports, comments, and commits to uncover their root causes. Our manual study involves three phases:

Phase I. Two authors of this study (A1 and A2) independently derived an initial list of the causes by manually inspecting the title, description, commit message, comment, and code change of each bug report.

Phase II. A1 and A2 unified the derived causes and compared the assigned cause for each database access bug. Any disagreement was discussed until reaching a consensus. The inter-rater agreement of the cause of bugs has a Cohen's kappa [46] of 0.83, indicating an almost perfect agreement [80]. To encourage replication of our results, we have made the dataset available online [10].

Phase III. We further grouped the database access bugs into five categories based on the stage when the bug occurs in the process of accessing the database: *SQL Queries*, *Schema*, *API*, *Configuration*, and *SQL query result*.

Results. Table 5 summarizes the manually-uncovered root causes of the database access bugs and the number of bug instances. Below, we discuss each category in detail.

Table 5: Categories of the root causes of database access bugs.

Root Cause	Description	# Bugs
SQL Queries		**169 (40%)**
SQL syntax error	The SQL query contains SQL syntax error, which causes runtime exceptions.	108
SQL logic error	The SQL query contains incorrect business logic (e.g., incorrect condition in the `WHERE` clause), where the returned results are not what the developers expected.	45
SQL query is incompatible with some DBMSs	The syntax of the SQL query is not compatible with some database management systems that the application promises to support.	11
Invalid user-defined function	The SQL query calls a user-defined function written in PL/SQL (i.e., stored procedure) which is missing or compiled with errors.	3
Error while converting data types	The SQL query fails when trying to convert from one data type to another.	2
Schema		**95 (22.5%)**
Violation of database constraint	The SQL query violates database schema constraints such as not-null, foreign key, and primary key constraint.	46
Non-existent table/column	The table/column specified in the SQL query does not exist in the database.	20

Continued on next page

Table 5 – continued from previous page

Root Cause	Description	# Bugs
Poor schema design	The design of the database scheme needs improvement or is incorrect.	11
Invalid/unexpected column value	The SQL query inserts or updates a table column with invalid/unexpected value (e.g., the size of the value is larger than the specified column size in the DBMS).	9
Invalid modification of schema	The SQL statement modifying the table schema is invalid (e.g., duplicate values exist before creating a unique index constraint on the table columns).	9
API		**92 (21.7%)**
Incomplete/invalid object values	Some fields of the entity object are not set or are set with invalid value when trying to save or update the object to the DBMS.	21
Incorrect flow of calling APIs	The application code calling the database access API does not conform to the expected flow. For example, the code to update an entity object never calls the save method after updating the object.	18
Inconsistent entity object state	Calling Hibernate APIs to modify entity objects whose states are inconsistent with the action (e.g., saving an entity object that is detached from Hibernate session, for which the change will not be reflected in the DBMS).	16
Bugs in database access APIs	There are bugs in the JDBC driver or the Hibernate framework.	9
Missing exception handling	There is no proper exception handling when calling database access APIs (e.g., missing try-catch block).	8
Incorrect parameter	The database access API is called with missing parameters or invalid arguments.	7
Hibernate proxy misuse	The Hibernate proxy object, representing a lazily loading object, is accessed incorrectly (e.g., accessing non-initialized fields of a proxy object directly).	6
Transaction misuse	Missing or using transaction incorrectly (e.g., missing annotation @Transactional when calling database access APIs).	4
Inefficient API call	The database access API call retrieves too much unnecessary data (e.g., retrieving all fields of an object while some of them are never used in the application).	3
Configuration		**38 (9%)**
Incorrect database connection	The configuration of the database connection is incorrect so that the connection to the DBMS cannot be established.	16
Incompatible database driver version	The database driver version is incompatible with the database version.	12
Incorrect ORM configuration	Bugs in ORM configuration, such as having a typo in ORM annotations that causes unexpected database access behavior.	10
SQL Query Result		**29 (6.9%)**
Incorrect entity object conversion	Database-returned results are converted to entity objects incorrectly (e.g., fields mismatch between the returned database record and the entity object).	15
Cache misuse	Developers use the cache of database records incorrectly (e.g., the cache is cleared unintentionally and causes performance bugs).	10
Missing cache	Developers do not add the needed cache for some database tables, which causes performance bugs.	4

- SQL Queries (169/423, 40%). Database-backed applications access DBMS data by issuing SQL queries (i.e., either manually constructed by developers or automatically generated by ORM frameworks) to DBMSs. Since the compiler cannot capture errors in the SQL query during compile time, any errors in the SQL query may return unexpected results or even runtime exceptions that may cause the application to crash. We find that, among all the bugs related to SQL queries, *syntax error in SQL queries* is the most common root cause (108/169, 63.9%). In this case, the SQL query issued to DBMSs violates the SQL syntax rule, causing runtime exceptions. These bugs usually happen when developers make a typo in the SQL query or fail to generate the criteria as expected in the SQL query. For example, `ADempiere` #2494 reports a runtime exception due to the syntax error as follows:

```
Query.list: SELECT Gender, ... FROM C_BPartner WHERE ( AND C_BP_Group_ID=103) AND ... [76]
org.postgresql.util.PSQLException: ERROR: syntax error at or near "AND"
```

The SQL keyword `AND` (as highlighted in red) violates the SQL syntax. `AND` should be used between two conditions and not immediately after the keyword `WHERE`. The SQL query was generated based on some developer-specified criteria to find specific records in the database table. The problematic SQL query was caused by some untested criteria, which resulted in generating incorrect logical operators (i.e., `AND`, `OR`, and `NOT`) to combine conditions in the `WHERE` clause.

The second most common root cause is *logic error in SQL queries* (45/169, 26.6%). Although the SQL query issued to DBMSs may be syntactically correct, the returned result may not be what the developers expected. These bugs usually happen when developers partially understand the business requirements of the underlying data query. For example, there may be a logical error (e.g., missing conditions) in the `WHERE` clause of the SQL query (e.g., `BroadleafCommerce` #586), which leads to unexpected query results. We also find cases where the *SQL queries are not compatible with some DBMSs* that the application promises to support (11/169, 6.5%). In some cases (3/169, 1.8%), we find that the *SQL query calls invalid user-defined functions*, which is missing (e.g., non-existent) or compiled with errors (e.g., `ADempiere` #828). Hence, the SQL query causes runtime exceptions, leading to no query results. Finally, in 2/169 (1.2%) cases, we find that there are *data conversion errors in SQL queries*. For example, in `Adempiere` #1174, the SQL query calls the `COALESCE()` function which fails to implicitly convert the data type from `VARCHAR` to `NVARCHAR2`. Developers fixed these bugs by using the target data type directly to avoid erroneous implicit conversion.

Overall, based on our manual observation, many SQL-related bugs may be revealed if developers have proper test cases in place. For example, if test cases cover all the SQL queries, then the test cases should be able to capture SQL syntax errors. Moreover, some logic errors in the SQL queries (e.g., the returned result is unexpected) may also be revealed if there are comprehensive test cases. After manually checking the test cases, we find that many problematic SQL queries are not

fully covered by test cases. One possible way to improve the quality of database access code is by introducing SQL query coverage as code coverage criteria to measure how well the database access code is tested. For researchers, our findings call for automatic test generation tools [119, 54, 33, 22] that can cover SQL queries in database access.

> Any errors in SQL queries issued to DBMSs may cause unexpected query results or even runtime exceptions. Our manual analysis finds that these problematic SQL queries are not fully covered by test cases. Future studies should emphasize the development of test generators that target SQL query coverage.

- Schema (95/423, 22.5%). The database schema defines how data is structured in the database (e.g., tables and data types) and how data records inside the database relate to each other (e.g., `foreign key`). When database-backed applications access the data, the DBMS receives the SQL query and executes it by querying the data defined by the database schema. Hence, any issues related to database schema may return unexpected results or cause runtime exceptions. We find that, among all the bugs related to the database schema, *violation of database constraint* is the most common root cause (46/95, 48.4%). In these cases, the SQL query violates one or more of the common database constraints: `foreign key`, `unique`, `not-null`, and `primary key` constraints, which causes an exception to occur. The violation of database constraints usually happens if developers do not handle corner cases or exceptions properly when they try to persist the data in database tables. As the database constraints are configured in the DBMS, the developers may not know their existence or fail to validate if the data complies with the database constraints in the application code. For example, `Openfire` #692 reports a runtime exception due to the violation of `not-null` database constraint when an SQL query tries to insert the value of `NULL` into a column that does not allow null value. The code snippet is shown as follows:

```
1  public class XMPPServer {
2      ...
3      try {
4          host = InetAddress.getLocalHost().getHostName();
5      } catch (UnknownHostException ex) {
6          Log.warn("Unable to determine local hostname.", ex);
7          host = "127.0.0.1";
8      }
9  }
10
11 public class DefaultSecurityAuditProvider {
12     ...
13     String LOG_ENTRY = "INSERT INTO ofSecurityAuditLog
           (msgID,username,entryStamp,summary,node,details) VALUES(?,?,?,?,?,?)";
14     PreparedStatement pstmt = con.prepareStatement(LOG_ENTRY);
15     ...
16     pstmt.setString(5, XMPPServer.getInstance().getServerInfo().getHostname());
17     pstmt.executeUpdate(); //execute the SQL
18 }
```

The value of `host` (i.e., hostname) is initialized in class `XMPPServer` (Line 4). In case `UnknownHost Exception` happens, developers try to assign an IP address (i.e., `127.0.0.1`) to the host name in the `catch` block (Line 7). Then, in class `DefaultSecurityAuditProvider`, developers construct the SQL query (Line 13) and set the parameter `node` with the value of host (Line 16). However, if an uncaught exception (any exception that is not `UnknownHostException)` occurs in class `XMPPServer`, the host name would be null and executing the SQL query in Line 17 would violate the `not-null` database constraint (the corresponding column of the table for `node` should not be NULL).

The second most common root cause is *non-existent table/column* in the database (20/95, 21.1%). Database-backed applications and the underlying database evolve during the software development, which may lead to inconsistency between the SQL query and database schema. For instance, the table/column specified in the SQL query may be deleted, renamed, or not yet created (e.g., `dotCMS` #4806) in the database. Another root cause is *poor schema design* (11/95, 11.6%), where the design of the database schema needs improvement or is incorrect, which may cause unexpected results. For example, missing a unique constraint on specific table columns may result in unintended duplicate table records on those columns (e.g., `dotCMS` #5755). We also find cases where the SQL query inserts or updates a table column with invalid/unexpected value (9/95, 9.5%). For example, the size of the value in the SQL query may be larger than the specified column size in the DBMS (e.g., `OpenMRS` #601). Finally, in 9/95 (9.5%) cases, we find that the SQL statement modifying the table schema is invalid because it violates how data is structured in the database or the constraint rule. For example, in `Adempiere` #1174 the `CREATE UNIQUE INDEX` statement fails to create the `unique` constraint as duplicate records already exist on the table columns.

We find that database access bugs caused by violation of database constraints usually happen if developers do not handle corner cases properly (i.e., untested conditions), which leads to an invalid value of persistent data that violates database schema constraints. For example, `NULL` value is invalid to be inserted into the database table if the corresponding column is configured with a *not-null* database constraint. The invalid value of persistent data may be detected earlier by test cases or avoided if developers have proper validation of the data before persisting it to the DBMS. We also find that the co-evolution of underlying database schema and code [125, 144] in database-backed applications may lead to violation of database schema in SQL queries (e.g., *non-existent table/column*). For example, the table name may still remain the same in SQL queries when it has been modified in the DBMS. Our findings suggest that developers should consider the underlying database schema when generating SQL queries and keep track of the database schema evolution to update the SQL query. Our findings also call for an automatic mechanism to maintain the consistency between SQL queries and database schema when the schema evolves.

SQL queries with invalid values of persistent data may violate database schema constraints and SQL queries may also violate database schema due to the database schema evolution, causing runtime exceptions. Future studies may provide support for the development and maintenance on SQL queries regarding database schema.

- API (92/423, 21.7%). In database-backed applications, developers call database access APIs (e.g., ORM APIs) to access database data. Bugs are introduced if developers partially understand the assumptions made by the rich set of APIs or use the API in a way that does not conform to the business logic of applications. While most issues are related to incorrect or inefficient usages of APIs (90.2%), there are still some issues related to the API itself (i.e., *bugs in database access APIs*, 9/92, 9.8%). We find that, among all the bugs related to database access API usage, *incomplete/invalid values in the entity object* is the most common root cause (21/92, 22.8%). These bugs usually happen when developers call database access APIs to persist the entity object with many fields, of which developers forget to set values of some fields (e.g., `OpenMRS` #3337) or set some fields with invalid values, resulting in incorrect records in database tables. The second most common root cause is *business logic error* when calling database access APIs (18/92, 19.6%). In this scenario, the application code calling the database access API does not conform to the application's business logic which may cause unexpected records in database tables. For example, in `BroadleafCommerce` #1538, the application intends to set the `payment` status in the database to *archived*. However, developers never explicitly call the database API to update the corresponding database record, which causes incorrect `payment` status. The third most common root cause is *inconsistent entity object state* when calling database access APIs (16/92, 17.4%). Developers may misuse the Hibernate session to modify the entity object, causing Hibernate exception `NonUniqueObjectException`, which means that the developer tries to associate two different entity objects with the same identifier value (i.e., primary key), in the scope of a single session. For example, the developer may try to save an entity object that is detached from Hibernate session, when a different object with the same identifier value is already associated with the session (e.g., `OpenMRS` #3728).

The next two most common root causes are *bugs in database access APIs* (9/92, 9.8%) and *missing exception handling* when calling database access APIs (8/92, 8.7%). For example, `DBeaver` #6554 reports unexpected query results caused by a bug in the JDBC driver for SQL Server (`mssql-jdbc` #969 [95]). We also find cases (7/92, 7.6%) where the database access API is called with *incorrect parameter*, missing parameters or invalid arguments (e.g., `null` argument in `BroadleafCommerce` #153). In some cases (6/92, 6.5%), we find that developers *misuse Hibernate proxy* where the Hibernate proxy, representing lazily loading object, is accessed incorrectly. The non-initialized fields of the proxy object are accessed directly by developers (e.g., `OpenMRS` #3340) instead of calling

the associated getter method which enforces Hibernate to initialize fields by querying the DBMS. In some cases (4/92, 4.3%), we find that developers *misuse the transaction*, which may cause data integrity bugs in the database (e.g., missing transaction in `BroadleafCommerce` #330). Finally, in (3/92, 3.3%) cases, we find that developers call the database API to retrieve too much unnecessary data, which may cause performance bugs (e.g., `BroadleafCommerce` #762).

We find that, when calling database APIs, many database access bugs are related to the discrepancy between database records and objects in object-oriented languages. Although ORM frameworks provide developers with a conceptual abstraction for mapping the database records to objects, any errors in entity objects may cause incorrect records in database tables (e.g., *incomplete/invalid values in the entity object*). We also find that developers may have difficulties managing the entity object when calling database access APIs (e.g., *inconsistent entity object state*) due to the complexity and impedance mismatches [41] of the object-relational mapping. Simplifying the complexity of database access APIs may help reduce database access bugs. Future studies may also provide support to developers in using database access APIs. For example, an entity object checker may help developers detect incomplete values in the entity object when calling database access APIs.

> When calling database APIs, many of the issues are related to incorrect API usage or API anti-patterns. Future studies may provide support for developers in using database access APIs.

- Configuration (38/423, 9%). In database-backed applications, developers need to set up various configurations (e.g., database connection) before accessing the database. Incorrect configurations may cause errors or unexpected behaviors when accessing the database. We find that, among all the database access bugs related to configuration, *incorrect database connection* is the most common root cause (16/38, 42.1%). The database connection is configured as a URL that contains information such as where to search for the database (i.e., the host name and port number of the node hosting the DBMS) and the name of the database to connect to. The information in the database connection URL may be incorrectly configured (e.g., `DBeaver` #9382), causing the connection error to the DBMS. The second most common root cause is *incompatible database driver version* (12/38, 31.6%), which may lead to database access failure. For example, some data types for a specific DBMS release may be changed and not supported by the database driver (e.g., `Openfire` #759), causing runtime exceptions. We also find cases where the *ORM configuration is incorrect* (10/38, 26.3%). Since the ORM frameworks automatically convert entity objects to/from the corresponding database record based on the configuration, incorrect configuration of ORM may lead to unexpected ORM framework behaviors. For example, in `BroadleafCommerce` #497, a duplicate annotation `JoinTable` is configured on the entity class `OfferCode`, which incorrectly adds additional duplicate records in the database table.

We find that developers may incorrectly set up configurations before accessing the database, which may cause errors or unexpected behaviors when accessing the database. We also find that developers may forget to update the configurations (e.g., database driver version) when the code or DBMS evolves. Our findings indicate that managing the configuration is a continuous process during the development and evolution of database-backed applications. Developers may benefit from tools that can detect incorrect configurations automatically. For example, an ORM configuration checker may help developers detect duplication annotations in ORM configurations.

> Incorrect configurations may cause errors or unexpected behaviors when accessing the database. Developers should develop automated tests that continuously verify various configurations. Future studies may also work on tools to help developers detect incorrect configurations automatically.

- SQL Query Result (29/423, 6.9%). Database-backed applications often convert the data records returned by the DBMS into objects in object-oriented programming languages. For frequently-queried data in the DBMS, developers also store the corresponding objects in the cache. We find that, among all the bugs related to SQL query results, *incorrect entity object conversion* is the most common root cause (15/29, 51.7%). These bugs are caused by incorrect conversion from database-returned results into entity objects and usually happen when developers mismatch the fields between them (e.g., `Openfire` #664), thereby causing inconsistency between entity objects in applications and database records. We also find cases where developers *misuse the cache* for database table records (10/29, 34.5%). For example in `dotCMS` #5553, developers incorrectly clear the entire cache instead of the corresponding cache needed after updating the data and saving it in the DBMS, which may cause performance bugs due to the unnecessary cache updates. Finally, in 4/29 (13.8%) cases, we find that developers did not use cache to store frequently-queried data in database tables (e.g., `dotCMS` #6288), which causes significant data retrieval overhead.

We find that database access bugs caused by *incorrect entity object conversion* usually occur when calling JDBC APIs to access the database. When using JDBC APIs, developers have to extract the SQL query results and convert the values to corresponding fields of the entity object. In contrast, when using ORM frameworks, the conversion is done automatically after developers configure the mapping between entity classes and database tables. While caching is a common way to improve the performance of database access, we also find that misuse of the cache may introduce performance bugs. Future studies may propose tools to help developers use caching frameworks when accessing the database [39].

When retrieving SQL query results, developers may incorrectly convert the database records returned by the DBMS into entity objects in applications, causing inconsistency between them. Developers may also misuse the cache which causes performance bugs, calling for tools to help developers use caching frameworks more intelligently when accessing the database.

3.3.3 RQ3: How do Categories of Database Access Bugs Prevail with Different Database Access Technologies?

Motivation. In RQ2, we analyzed the root causes of database access bugs and grouped them according to several categories. In general, database accesses leverage two widely used technologies: (i) SQL queries (e.g., JDBC), and (ii) ORM (e.g., Hibernate). These database access technologies have unique design principles and goals. For example, rather than constructing SQL queries in the database access code, ORM enables developers to manipulate persistent data as if it is in-memory objects [150]. Thus, studying bugs related to these two technologies allows us to better understand their maintenance challenges. In this RQ, we investigate how different categories of database access bugs prevail with these two technologies. Our findings may guide future studies to provide support for developers in maintaining database-backed applications that use different technologies.

Approach. We further analyze the database access bugs that we studied in RQ2 and classify the bug reports by the technology (i.e., JDBC or Hibernate) used to access the DBMS. We perform a manual inspection of the source code associated with the bug fix to verify the usage of the technologies. For each bug category, we compare the percentage of database access bugs related to each technology.

Results. Among all the studied applications, `metasfresh`, `Openfire`, `ADempiere`, and `DBeaver` use JDBC while `BroadleafCommerce`, `dotCMS`, and `OpenMRS` use Hibernate to access the database. Accordingly, 237 database access bugs are related to JDBC, and 186 are related to Hibernate. Figure 3 compares the distribution of database access bug categories between JDBC and Hibernate. We find that *SQL queries*, *Schema*, and *API* bugs cover (356/423, 84.2%) database access bugs.

- SQL queries bug (128/237, 54%) is the most frequent issue when using JDBC, while API bug (72/186, 38.7%) is the most frequent issue when using Hibernate. The possible reason is that developers manually construct the SQL queries when using JDBC, while Hibernate generates the SQL queries automatically which makes it less prone to errors in SQL queries (e.g., SQL syntax errors). In terms of API bugs, the advanced Hibernate features (e.g., Hibernate session and Hibernate proxy) make Hibernate APIs more complex to use compared to JDBC APIs. We find that, when using Hibernate, most of the API bugs happen when developers call database access APIs to persist the entity objects (e.g., updating the value of the entity object to the corresponding

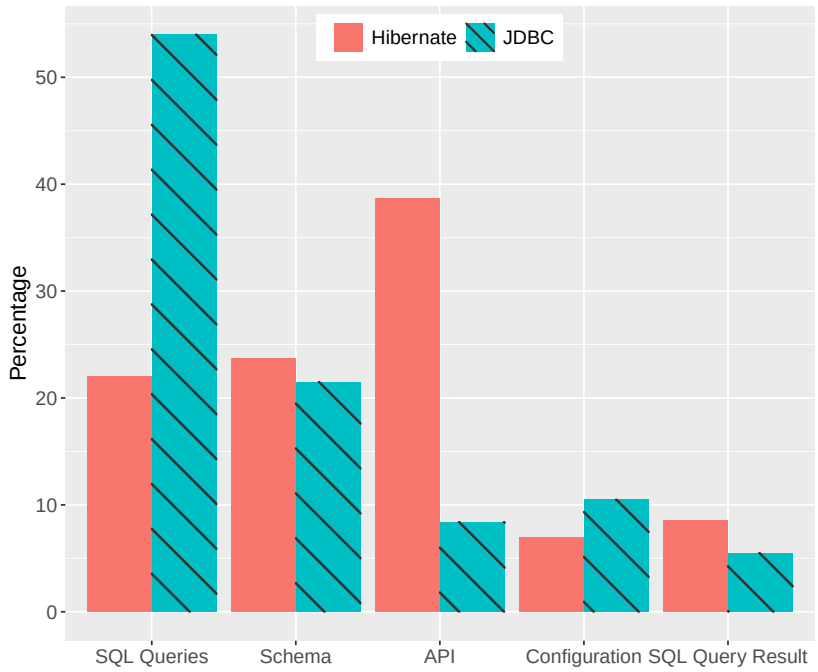


Figure 3: Distribution of the categories of database access bugs that occur in JDBC and Hibernate database-backed applications.

database record). Our findings suggest that developers should pay more attention to constructing the SQL queries when using JDBC and pay more attention to API usage, especially persistent APIs, when using Hibernate.

- There are many SQL query bugs (128/237, 54%) when using JDBC, while there are still SQL query bugs (41/186, 22%) when using Hibernate. As discussed in RQ2, most of the SQL query bugs are caused by *syntax error in SQL queries*. Although Hibernate automatically generates the SQL queries executed by DBMSs, it provides APIs to use Hibernate Query Language (HQL) queries [68]. Hibernate may also generate problematic SQL queries if there are errors in HQL queries manually constructed by developers (e.g., `OpenMRS` #5359). Hence, using Hibernate may still result in SQL query bugs.

- There are schema bugs when using JDBC (51/237, 21.5%) and Hibernate (44/186, 23.7%). 1) When using JDBC, schema bugs are mainly caused by *Non-existent table/column* (18/51, 35.3%), mostly due to the co-evolution of underlying database schema and code (as discussed in RQ2). In contrast, schema bugs caused by *Non-existent table/column* only account for (2/44, 4.5%) when using Hibernate. The possible reason is that, when database schema evolves (e.g., the table column name changes), developers only need to modify the mapping between entity/table once when using Hibernate, but have to manually modify all corresponding SQL queries when

using JDBC. Developers using JDBC may benefit a lot from automatic tools to help maintain the consistency between SQL queries and database schema. 2) When using Hibernate, schema bugs are mainly caused by *violation of database constraint* (31/44, 70.5%). Hibernate provides built-in constraints [67] on entity fields to help developers prevent *violation of database constraint*. For example, the annotation `@NotNull` in Hibernate declares a field to be not-null. If the field value is null, Hibernate will not execute any SQL statements and prevents storing null values in the underlying database, which avoids the violation of database constraint `not-null`. However, Hibernate does not provide corresponding built-in constraints for other database constraints such as `foreign key` (e.g., `BroadleafCommerce` #678), `primary key`, or `unique constraint`. Developers of Hibernate framework may provide more built-in constraints in the future to help developers deal with common database constraints [154].

- There are cache issues in SQL query result bugs when using JDBC (5/13, 38.5%) and Hibernate (9/16, 56.3%) We find that, when using JDBC and Hibernate, some SQL query result bugs are related to the cache, which may cause performance issues. Future studies of performance in database-backed applications may address SQL query results when accessing the database.

SQL queries, Schema, and API bugs cover 84.2% of database access bugs across all studied applications. SQL queries bug (54%) and API bug (38.7%) are the most frequent issues when using JDBC and Hibernate, respectively. Hibernate cannot abstract database access completely, so many issues such as SQL query bugs and Schema bugs still exist when using Hibernate. Future studies should provide support for developers in using different database access technologies.

3.4 Discussion

We now state the actionable implications of our findings and highlight opportunities for future work.

There is a need for better support for the development and maintenance of database access code in database-backed applications. In RQ2 and RQ3, we find that bugs related to SQL queries or the database schema (e.g., syntax error or inconsistency with the database schema) are the most frequent category of database access bugs when using JDBC. For example, developers may make a typo in the SQL query or fail to construct the search criteria as expected in the SQL query, leading to problematic SQL queries which are not checked at compile time. We also find that this type of bug still exists when using Hibernate, since developers may need to use JPQL for more complex database access. Therefore, to assist developers with improving the quality of database-backed applications, there is a need for better tooling support to verify the SQL queries and database

schema. For example, future research may work on tools that statically verify the syntax of SQL queries and the consistency between database schema and the SQL queries in the application code. Although there are some static analysis tools, such as `dbcritic` [6] and `holistic` [7], that try to help detect SQL schema issues using static analysis, these tools can only analyze specific database access frameworks (mostly only JDBC) or DBMS due to the limitation of static analysis. Building more generic static analysis tools that can detect errors in SQL queries will alleviate more SQL query bugs during system development. Future research may also work on approaches to automatically maintain database access code. Developers may benefit from approaches that automatically suggest updates to the database schema or database access code when developers modify the code – change impact analysis. For example, when a developer modifies the database schema, the approach can automatically identify all the database access code (e.g., either SQL queries or ORM database access APIs) that is impacted by the change and require an update.

Future studies should help developers better leverage ORMs such as Hibernate. In RQ2 and RQ3, we find that API bug is the most frequent issue when using Hibernate. Compared to JDBC APIs, the advanced Hibernate features (e.g., Hibernate session and Hibernate proxy) make Hibernate APIs more complex to use. When calling Hibernate APIs, developers may persist entity objects with errors (e.g., *incomplete/invalid object values*), which causes incorrect records in database tables. While there are many prior studies [38, 150, 151, 128, 85] focus on helping developers detect performance issues when using ORM APIs, there is limited study on the functional aspect. Future studies should provide support for developers in using Hibernate APIs. For example, future research may propose an entity object checker that helps developers detect incomplete values in the entity object before calling Hibernate APIs. Another possible idea is to check whether every field of the persistent entity has been initialized or set with a value. This helps ensure the validity of the database-managed objects and avoid runtime errors. We also find that there are many issues related to the incorrect flow of calling APIs or API parameters. Future studies can also propose approaches to help detect issues in ORM API usage.

Designing and Generating adequate test cases for database access code. In RQ2, we find that most database access bugs cause severe problems like unexpected query results or runtime exceptions that may cause the application to crash. Our manual study finds that these bugs are usually not fully covered by test cases. For example, SQL queries with invalid values (e.g., `NULL`) of persistent data may violate database schema constraints (e.g., *not-null*). These bugs occur if invalid values are generated from untested conditions. In order to improve the quality of database-backed applications, developers may consider coverage of both SQL queries (syntax and semantics) and database access API usage to measure how well the database access code is tested. Developers may also consider different types of coverage such as database coverage (e.g., coverage based on database

schema, constraints, or database access code) since databases are the key components of database-backed applications. Future studies may consider this coverage as code coverage criteria in automatic test generation tools for database-backed applications. For example, future studies may use database coverage to guide test case generation using search-based or fuzz-based approaches. Future studies may also propose metrics or approaches to evaluate the effectiveness of the existing test cases. For instance, there may be a need to design specialized mutation operators for database-related tests and help developers with the quality assurance of database-backed applications.

Complementary to developers in selecting database access technologies. Developers often rely on two main technologies (i.e., SQL queries and ORM frameworks) to access the underlying database and discuss a lot on how to select them. We select the top 15 questions that compare Hibernate to JDBC in Stack Overflow and manually examine the answers. We find that developers often make trade-offs in the selection, mainly focusing on the strengths and limitations between JDBC and Hibernate. For example, one strength of using Hibernate is that developers do not need to write SQL queries (*"Such ORMs provide the maximum level of abstraction to the point you almost never have to write SQL queries."* [9]). On the other hand, no explicit SQL in the source code when using Hibernate sometimes makes debugging and performance tuning difficult (*"The time savings gained are easily blown away when you have to debug abnormalities resulting from the use of the ORM."* [8]). We also contacted the main contributors of the studied applications by inquiring why and how they selected JDBC or Hibernate in their applications. They mention that Hibernate makes their code easy to understand and modify since the application is open source and has contributors at every level from around the world. However, they also address that developers may not truly know the automatically generated SQL queries by Hibernate, which may cause major slowdowns or failures after deployment. Our findings provide complementary to developers in selecting database access technologies. For example, considering developers' capabilities in technologies, if they have a good understanding of the entity object in Java code, they may make fewer bugs (i.e., database access bugs related to *incomplete/invalid values in the entity object* or *inconsistent entity object state*) when calling Hibernate APIs.

3.5 Threats to Validity

External Validity. One possible threat to external validity is the generalization of the dataset (i.e., database access bugs) we collected. To ensure that the applications we study are large enough and well maintained, we apply three criteria to select the studied Java applications based on the number of stars, the use of issue tracking systems, whether containing database-related bugs, and active maintenance activities (having more than 1,000 commits and at least 100 fixed bug reports). Based

on these criteria, we ended up with seven applications, all of which use Hibernate or JDBC to access the DBMS. These applications contain thousands of lines of code (117K $\sim$ 1,610K), thousands of commits (10K $\sim$ 52K), and a pronounced development period (5 $\sim$ 16 years), across different domains such as e-commerce, ERP, and database tools. We do notice that metasfresh is a fork of ADempire, but the fork was done in 2015 due to the development gap compared to the latest ADempiere codebase[2]. Since then, both applications have grown apart and there has been active development on metasfresh. As shown in Table 2, the studied periods of both applications are from after 2015, which cover the multiple years of development activities and database access bugs after the fork. The LOC also differs significantly between metafresh and ADempiere (879K vs. 1,610K). Therefore, due to the difference in development activities, we include these two applications in our study. Developers can also use these database access frameworks to interact with a wide range of underlying relational DBMSs (e.g., MySQL, PostgreSQL, or Oracle). There may be some framework-specific issues when using other database access frameworks, but many of the issues that we found in this study are not specific to one framework (e.g., there are many bugs related to SQL query syntax or database schema). Therefore, we believe our findings offer valuable insights for both researchers and practitioners. We acknowledge that non-relational databases such as NoSQL are becoming more popular. However, in this study, we focus on studying the systems that use relational database management systems (e.g., MySQL or PostgreSQL). This focus is due to their established history and the relative scarcity of studies examining the characteristics of database access bugs in these environments. Future studies should consider NoSQL as it is also a critical topic, especially given the variety of NoSQL database vendors and data types (e.g., graphs or documents).

Internal Validity. The main threat to the internal validity of our results could be the bias when deriving bug causes. To mitigate this threat, two authors of this study (A1 and A2) independently inspected the bug report and commit to each bug to identify the cause. The inter-rater agreement of the cause of bugs was measured using Cohen's Kappa coefficient and the disagreements were discussed until reaching a consensus. We considered only the database access technologies (i.e, JDBC and Hibernate) as the factors for the categories of database access bugs. However, other factors (e.g., design, coding style, and framework) may also affect the design of database access code. To mitigate this threat, we carefully examined the documentation and source code of the studied applications and found that they mainly use JDBC or Hibernate directly to access the database. Therefore, we believe that other factors should not significantly affect the database access and thus the occurrence of categories of database access bugs. We only conducted our study in Java applications, but there are other database-backed applications implemented in other programming languages such as Python and Ruby. These programming languages also have various ORM frameworks available (e.g.,

[2]https://en.wikipedia.org/wiki/Metasfresh

Django, and Ruby on Rails) that may have their own unique challenges. Based on the literature [38, 41, 39, 151, 150], there are many overlaps in the database access performance problems between Ruby and Java. Common issues in both Ruby and Java include retrieving more data than needed, not using batching for database access, caching, and inefficient ORM API usage. Therefore, we expect that there are some similar issues in the database-backed applications that are implemented in other programming languages, but there should also be language- and framework-specific issues. Future studies are needed to further study the issues in applications that are implemented in other programming languages and identify commonalities and differences between them.

Construct Validity. The construct validity of our study rests on the methodology of collecting database access bugs from the fixed bug reports of studied applications. First, for applications with fixed bug reports of more than 1,000, we conduct the study on a statistically significant sample from all fixed bug reports randomly under a 95% confidence level and a 3% margin of error, which may introduce minimal noise. Second, we filtered the studied bug reports by searching for database-related keywords in each bug report. The keywords were derived based on manual analysis and hence, may not be comprehensive. We may have omitted seemingly trivial database-related bugs with very little reported information. The similarity in the trends between the database and non-database bugs may be related to the application's development process. Hence, we manually examine the code repositories, release notes, and development history to uncover the possible development process that the applications follow. Overall, we find that all the studied applications have adopted continuous integration and agile development, at least since the past decade. Most of the applications, such as DBeaver and dotCMS, have a consistent release cycle of three months. We further examine the spikes in the number of reported issues as shown in Figure 2, and we find that most spikes are related to having more code changes (e.g., a major release). In short, we do not find a clear connection between the software development process and the trends across the studied applications. On the other hand, we find specific reasons for some database access bugs (as discussed in RQ2). For example, `ADempiere` #2494 reports a runtime exception due to the syntax error in the SQL query. The SQL query was generated based on some developer-specified criteria to find specific records in the database table. The problematic SQL query was caused by some untested criteria, which resulted in generating incorrect logical operators (i.e., AND, OR, and NOT) to combine conditions in the WHERE clause. Another example is `Openfire` #692 which reports a runtime exception due to the violation of not-null database constraint when an SQL query tries to insert the value of NULL into a column that does not allow null value. The bug happens when developers do not handle corner cases properly (i.e., untested conditions). In short, these bugs may happen at any stage of the development when developers modify existing database access code, fix other bugs, or

add new features. However, our finding shows that database access code is the core of database-backed applications. Although these applications may have well-designed databases, database access code co-evolves with other source code, requiring continuous attention for maintenance and quality assurance.

3.6 Related Work

In this section, we discuss the related work of our study.

Database access issues in database-backed applications. Prior works study the database access issues in database-backed applications from different perspectives, e.g., syntactic or semantic errors in SQL queries [28, 17], SQL anti-patterns [77, 24, 52, 18], SQL code smells [126, 129, 100], and performance issues [38, 150, 151, 128]. Specifically, Brass and Goldberg [28] proposed a list of semantic errors in SQL queries. The book by Karwin [77] provided an overview of SQL design anti-patterns containing four categories: logical database design anti-patterns, physical database design anti-patterns, query anti-patterns, and application development anti-patterns when employing SQL in the application code. A recent survey by Alshemaimri et al. [18] summarized categories of SQL anti-patterns and framework-specific (e.g., ORM) anti-patterns. Redgate [126] documented 119 SQL code smells, concerning database design issues, table design, data types, expressions, naming, routines, query syntax, and security loopholes. Shao et al. [128] conducted a literature survey and reported 34 database access performance anti-patterns in total.

Despite these efforts, there is a lack of study toward understanding database access bugs using SQL queries or ORM frameworks. Database access bugs in database-backed applications during runtime may be different from the syntactic or semantic errors in separate SQL queries since database access leverages SQL queries embedded within the application code or generated by the ORM frameworks to interact with DBMSs. On the other hand, unlike SQL anti-patterns or SQL code smells, which allow programs to execute correctly but have quality problems such as poor performance or indicate the presence of quality problems but not necessarily bugs, respectively, database access bugs may cause severe problems like crashes. Our work addresses this gap by providing categories of database access bugs from the issue tracking system and highlighting their root causes. We believe that the category and our actionable implications would help developers understand the maintenance issues and challenges in database-backed applications.

Database schema and program co-evolution. Prior research [125, 42, 49, 89] study the co-evolution between database schemas and database-backed application programs. Curino et al. [49] studied the database schema evolution on Wikipedia and the effect of schema evolution on the system front-end. Maule et al. [89] proposed a program analysis-based approach to perform change

impact analysis on applications caused by database schema changes. Qiu et al. [125] conducted a comprehensive empirical analysis of how programs co-evolve with schema changes in database-backed applications. They find that database schemas evolve frequently and schema changes induce significant code-level modifications. Chen et al. [42] reported that in particular ORM-related code, changes are more scattered and frequent than regular code and these changes mostly address performance or security concerns. Previous studies mainly focus on how database schema changes impact applications programs, while we investigate how they may cause bugs in applications. In our work, we generalize categories of the root causes of database access bugs, of which some bugs are caused by database schema changes. Our derived categories provide a finer-grained view of the maintenance challenges that developers encounter.

3.7 Conclusion

In this chapter, we conducted an empirical study of 423 database access bugs from seven real-world Java database-backed applications. We find that although the number of reported database and non-database access bugs share a similar trend, their modified files in bug fixing commits are different. This implies that they are not necessarily co-located and database access bugs may have different causes as compared to non-database access bugs. Execution of SQL queries forms an integral part of database accesses and weighs 40% (169/423) of the studied bugs. The downside is that errors in SQL queries are not checked at compile time, thereby leading to erroneous access to the database. Further, many problematic SQL queries are not fully covered by test cases. This suggests that future development of test generators for database-backed applications should target SQL query coverage. Additionally, databases are the key components of database-backed applications and bugs related to database schema weigh 22.5% (95/423) of the studied bugs. This calls for the development of test cases that comprehensively cover the database (e.g., database constraint). We believe that our findings would provide developers with a comprehensive understanding of database access and aid related research on bug detection, testing, and debugging regarding database access.

Chapter 4

Localizing the Origin of SQL Queries in Database-Backed Web Applications

In database-backed web applications, developers often leverage Object-Relational Mapping (ORM) frameworks for database access. ORM frameworks provide an abstraction of the underlying database access details so that developers can focus on implementing the business logic of the application. However, due to the abstraction, developers may not know where and how a problematic SQL query is generated in the application code, causing challenges in debugging database access problems. In this chapter, we propose an approach, called SLocator, which locates where a SQL query is generated in the application code. SLocator is a hybrid approach that leverages both static analysis and information retrieval (IR) techniques. SLocator uses static analysis to infer database access for every possible path in the control flow graph. Then, given a SQL query, SLocator applies IR techniques to find the control flow path (i.e., a sequence of methods called in an interprocedural control flow graph) whose inferred database access has the highest similarity ranking. We implement SLocator for Java's official ORM API specification (JPA) and evaluate SLocator on seven open source Java applications. We find that SLocator is able to locate the control flow path that generates a SQL query with a Top@1 accuracy ranging from 37.4% to 70% for SQL queries in sessions, and 30.7% to 69.2% for individual SQL queries; and Top@5 ranging from 78.3% to 95.5% for SQL queries in sessions, and 59.1% to 100% for individual SQL queries. We also conduct a study to illustrate how SLocator can be used to locate issues in database access code.

An earlier version of this chapter is published at IEEE Transactions on Software

Engineering, vol. 49, no. 6, pp. 3376-3390, 1 June 2023. [83]

4.1 Introduction

Modern database-backed web applications are becoming more complex due to the ever-increasing functionality. To reduce development efforts and allow developers to focus on the business logic of the applications, database-backed web applications often use Object-Relational Mapping (ORM) frameworks to abstract database accesses. ORM frameworks have become increasingly popular with implementations in most modern programming languages such as Java, C#, Python, and Ruby [42, 151]. A report also shows that among the 2,164 surveyed Java developers, ORMs are the leading means of database access and 67.5% use Hibernate (one of the most popular Java ORM frameworks) instead of other database abstraction frameworks [30]. ORM provides a conceptual mapping between objects in object-oriented programming languages, such as Java, and tables in database management systems (DBMSs). With ORM mapping, developers can access the DBMS through a combination of object modifications and ORM API calls. For example, by calling `user.setName("Alice")` followed by `entityManager.persist(user)`, the ORM framework would automatically generate a SQL query, such as `UPDATE User SET userName = "Alice" WHERE ...`, which updates the user name in the DBMS.

Due to their intuitive abstraction of database access, ORM frameworks are widely used in database-backed web applications [153, 40, 38]. Despite the popularity and convenience of ORM frameworks, they may also cause maintenance challenges [42]. ORM automatically generates SQL queries based on various ORM configurations (e.g., the relationship among object types) and the called ORM APIs. As a result, developers do not have direct control over how the SQL queries are generated by ORM. When there are issues with a generated SQL query, developers may have difficulties knowing how the SQL query is generated and where in the code [153, 31, 41].

Hibernate, which is one of the most popular Java ORM frameworks, provides a mechanism that allows developers to record the generated SQL queries [70], i.e., ORM logs. Such ORM logs comprehensively record every generated SQL query so that developers would know what the generated queries look like. However, even with the recorded SQL queries, it would be difficult to infer how and locate where a given SQL query is generated [40, 106]. Hibernate generates a SQL query by considering all of the ORM configurations (e.g., how objects should be retrieved from the DBMS), how the objects are accessed, and the executed ORM APIs, on one code path. There may be hundreds or even thousands of database accesses in the source code. Thus, simply searching for the query text in the code would not work, and the generated SQL query may change based on the ORM configuration and the executed branch on an execution path.

Prior studies [158, 147, 149, 142, 146, 82, 123, 35] propose information retrieval based bug localization (IRBL) approaches that try to locate buggy files given some software artifacts (e.g., bug reports). IRBL approaches compute and rank the files based on their similarity with the given software artifact, where the files with the highest similarity are more likely to be defect-prone. IRBL approaches provide good indications of where the bugs are given limited information of the bugs [99]. Similarly, the ORM-generated SQL queries may have quality issues that are caused by incorrect or inefficient usage of ORM code/configuration [38, 150, 128].

In this study, we propose an approach, SLocator, that combines static analysis and information retrieval techniques to locate the origin (i.e., the control flow path, which contains a sequence of method calls) that generates a given SQL query in database-backed web applications. Different from prior studies on database-backed applications [151, 40, 38, 150], which focus on statically detecting issues based on predefined/known anti-patterns in database-backed web applications, SLocator can be used to locate the origin of any given SQL queries. SLocator also complements existing studies, which rely mostly on static analysis, by providing an approach to help analyze the dynamically-generated SQL queries.

SLocator is a hybrid approach that combines both static code analysis and information retrieval techniques for localization. First, SLocator applies static analysis to analyze the ORM configurations in the source code, which specify the mapping between objects and database tables, and how various objects should be retrieved from the DBMS. Then, SLocator statically analyzes each web request handling method and constructs interprocedural control flow graphs. For each path in the control flow graph, SLocator analyzes both the called ORM APIs and the ORM configurations to statically infer the database access (i.e., templated SQL query). Given a SQL query recorded by a DBMS, SLocator pre-processes the query to remove dynamic elements (e.g., dynamically generated values). Finally, SLocator uses cosine and Jaccard distance to find the control flow path for which the inferred database access has the highest similarity ranking with the given SQL query. Different from existing IRBL approaches, SLocator locates the control flow path that generates a given SQL query instead of the file/method that contains the corresponding ORM code. We choose to locate the control flow path because prior studies found that control flow paths provide additional information for locating the root causes of an issue [35, 26, 155, 134]. Moreover, database access issues may not only exist in ORM API calls but may also be related to how the objects are accessed during execution and the corresponding ORM configuration [38, 41, 150].

We implement SLocator for the Java Persistent API (JPA), which is the official ORM API specification for Java. We evaluate SLocator on seven open source database-backed web applications which use the Hibernate ORM framework. SLocator uses DBMS logs (e.g., MySQL logs) as the input. We use DBMS logs instead of ORM logs because DBMS logs are lightweight and commonly

used in production to record problematic SQL queries. In contrast, ORM log introduces significant performance overhead [69, 156, 159], as ORM would record every executed SQL query. Since large-scale web applications may execute hundreds of SQL queries per second, such performance overhead makes enabling ORM logs impractical in production. The dataset of SLocator is publicly available [11].

The main contributions of this study are:

- SLocator is one of the first techniques that combine interprocedural control flow analysis and information retrieval techniques for localization.
- SLocator is able to locate the control flow path that generates a given set of SQL queries with high accuracy (average Top@5 is 88.8%).
- We evaluate SLocator on existing problematic SQL queries (i.e., slow SQL queries) and we find that SLocator can locate where the SQL queries are generated with similarly high accuracy.
- We conduct a study to illustrate how SLocator helps locate slow SQL queries and database deadlocks in studied applications.

In conclusion, our study proposes a novel approach that is able to locate the control flow path that generates a given SQL query. Our research also illustrates the potential direction of leveraging static code analysis to enhance software artifact/bug localization techniques.

Chapter Organization. The rest of the chapter is structured as follows. Section 4.2 introduces the background of using ORM in database-backed web applications and surveys related work. Section 4.3 presents our approach in detail. Section 4.4 evaluates our approach on seven open source applications and conducts a study on locating the origin of problematic SQL queries. Section 4.5 discusses threats to validity. Finally, Section 4.6 concludes the chapter.

4.2 Background and related work

In this section, we first provide some background knowledge of ORM frameworks. Then, we use an example to illustrate the challenge of manually locating the origin of an SQL query (i.e., the control flow path that generates the query). Finally, we discuss related work in three areas: quality assurance of database-backed web applications, SQL query extracting, and IR-based bug localization.

Background of ORM. Object-relational mapping (ORM) frameworks provide a conceptual abstraction between objects in object-oriented languages and the data stored in the underlying DBMS [37].

Pet.java	Owner.java
@Entity @Table(name = "pets") public class Pet { @Id @Column(name = "id") private Integer id; @ManyToOne private Owner owner;	@Entity @Table(name = "owners") public class Owner { @Id @Column(name = "id") private Integer id; @OneToMany(fetch = FetchType.EAGER) @Fetch(value = FetchMode.JOIN) private Set<Pet> pets;

(a) Entity mapping.

```
public Owner findById(Integer id) {
    return entityManager.find(Owner.class, id);
}
```

id = 1

```
select owner0_.id as id1_0_0_, ... pets1_.id as id1_1_1_, ... from owners owner0_
left outer join pets pets1_ on owner0_.id=pets1_.owner_id where owner0_.id=1
```

(b) Translating objects to SQL queries by ORM.

Figure 4: An example of accessing the DBMS using ORM.

To leverage ORM frameworks, developers need to first specify ORM configurations. ORM frameworks have two main types of configurations. The first type of the configuration is the mapping configuration, where developers configure the mapping between entity classes and database tables. As shown in Figure 4a, the two entity classes, Pet and Owner (annotated with *@Entity*), are mapped to the pets and owners tables in the DBMS, respectively, using the annotation *@Table*. Both Pet and Owner entities have primary keys (*@Id*) which are mapped to database columns named id (*@Column*). Such mapping configuration allows ORM frameworks to automatically convert an object to/from the corresponding database record.

The second type of configuration relates to entity relationship and data retrieval strategies. ORM provides annotations that allow developers to specify the entity relationship to represent the business logic. For example, Pet has a *@ManyToOne* relationship with Owner, meaning that multiple pets may belong to the same owner. Similarly, there are *@OneToOne*, *@OneToMany*, and *@ManyToMany* relationships. The relationship between entity classes affects how ORM frameworks retrieve the corresponding object from the DBMS. By default, objects with *@OneToOne* and *@ManyToOne* relationship are retrieved together (i.e., eager retrieval), while objects with *@OneToMany* and *@ManyToMany* are not retrieved at the same time (i.e., lazy retrieval) for performance

optimization reasons [111, 78]. Developers can also explicitly specify the retrieval strategy. For example, by adding *FetchType.EAGER* to the configuration of Owner, as shown in Figure 4a, ORM will retrieve all the associated pets at the same time regardless of the type of the relationship. Finally, developers can configure how the associated objects are retrieved. For example, a *FetchMode.JOIN* to the configuration of Owner configures ORM to use an outer join to load the associated pets when fetching.

Despite ORM's advantages in abstracting database access, various configuration options and the paths that the application takes may affect how the SQL query is generated. Therefore, it may cause challenges in locating the origin of the SQL query. For example, as shown in Figure 4b, the ORM API, entityManager.find, retrieves the Owner object from the DBMS based on the owner ID. The corresponding SQL query generated by ORM retrieves not only the owner data but also the associated pets at the same time using an outer join according to the configured ORM annotation. The ORM-generated SQL query does not explicitly exist in the source code and the dynamically generated aliases (e.g., *owner0_* and *pets1_* for the owners and pets tables) introduce discrepancy when locating the origin of a SQL query. Therefore, if there are issues with a generated SQL query, it is challenging to locate where the SQL query is generated when diagnosing the database access code. Moreover, due to the complexity of database-based web applications, different code paths with the same root may generate slightly different SQL queries based on different API calls, which further increases localization difficulty. In short, manually locating where a SQL is generated can be time-consuming and challenging, especially given the size of modern database-backed web applications.

Below, we discuss related work relevant to this study.

Quality assurance of database-backed applications. Most prior research aims to study and detect performance issues in database-backed applications that are developed using ORM. Yan et al. [150] studied database-related performance inefficiencies in real-world web applications that are built using the Ruby on Rails ORM framework. They concluded several performance anti-patterns and proposed detection algorithms based on static analysis [151]. Shao et al. [128] presented a comprehensive empirical study that characterizes performance anti-patterns related to database accesses in web applications. Brass and Goldberg [28] summarized common SQL anti-patterns and how to address them. Chen et al. [40, 38] proposed an automated framework to detect and prioritize both performance and functional ORM anti-patterns. Grechanik et al. [62] proposed a run-time monitoring technique to detect database deadlocks.

Most prior studies focus on detecting database access issues at the code level (i.e., anti-patterns). Anti-patterns in ORM code may lead to generating inefficient or incorrect SQL queries. However, one limitation of the prior approaches is that they focus on detecting issues based on predefined/known

anti-patterns [128] in database-backed web applications and cannot detect issues that do not belong to any of the predefined anti-patterns. In contrast, given a potentially problematic SQL query (e.g., slow SQL queries or SQL queries that cause database deadlocks), our approach locates the code path that generates the query. Hence, we complement prior approaches by identifying the code that may result in generating problematic SQL queries.

SQL query extracting. Table 6 summarizes the related studies that perform SQL query extracting statically from the source code in their work. The closest related work is by Nagy et al. [106] which is the only study focusing on locating SQL queries. The authors proposed a static concept location approach to match HQL/JPQL query string in the code and the generated SQL query by comparing their abstract syntax trees (AST). However, they do not consider ORM APIs to access entity objects or ORM configurations. In our work, we consider not only static SQL queries (i.e., JPQL) but also ORM API calls to access entity objects and ORM configurations. When using ORM APIs to access entity objects, many SQL queries are generated dynamically, so it is not possible to locate where they are generated by doing string matching. In addition, we locate the control flow path (CFP) that generates a given SQL query instead of the method that contains a database access call (e.g., where the SQL query is defined). Prior studies show that such CFP provides important information when locating a fault and diagnosing the issues [35, 26, 155]. We find that our approach can locate the control flow path that generates a given SQL query with high accuracy. A relevant tool in this context is Hypersistence Utils [96], which logs SQL queries generated by Hibernate along with the stack traces of their associated methods. While it helps developers trace SQL queries back to the application code, it requires integration into the application and configuration to monitor specific packages. Furthermore, it introduces significant performance overhead by logging every query and method, making it impractical for production environments in large-scale web applications with high query volumes. In contrast, our static analysis tool eliminates the reliance on runtime logging, making it a more practical and efficient solution for production use.

Other studies on database-backed applications address different issues. Prior studies focus on detecting ORM code smell [71], conducting empirical studies of how SQL queries are constructed [19], extracting SQL queries from source code [93, 88, 105], checking the correctness of SQL queries [61], analyzing SQL queries [20], or detecting SQL anti-patterns [86]. Many studies [19, 88, 61, 20] do not support ORM frameworks but statically extract embedded SQL queries manually constructed by developers from the source code. Some studies [71, 93] partially supported ORM frameworks by extracting SQL queries from the source code. However, the extracted SQL queries are different from the dynamically generated SQL queries by the ORM during runtime, which still leaves the task of locating SQL queries challenging.

Information retrieval based bug localization. Information retrieval based bug localization

Table 6: Related studies that perform SQL query extracting statically from the source code. ORM, JPQL, and ORM APIs to access entity objects indicate whether the SQL query extracting supports ORM frameworks, JPQL, or ORM APIs to access entity objects, respectively.

Study	Summary of study	Goal of study	ORM	JPQL	ORM APIs to access entity objects
Nagy et al. [106]	They proposed a static concept location approach to match HQL/JPQL query string in the code and the generated SQL query by comparing their abstract syntax trees (AST).	Locating SQL	Yes	Yes	No
Huang et al. [71]	They proposed a static analysis tool, called HBSniff, for detecting 14 code smells.	Detecting ORM code smell	Yes	Yes	No
Anderson [19]	They studied five patterns of SQL query construction in actual PHP systems.	Empirical study of SQL	No	N/A	N/A
Meurice et al. [93]	They presented a static analysis approach to extract SQL queries in Java systems.	Extracting SQL	Yes	Yes	No
Manousis et al. [88]	They presented a method that identifies the embedded queries within database applications.	Extracting SQL	No	N/A	N/A
Nagy and Cleve [105]	They briefly described the tool that is able to extract SQL queries from Java code through static string analysis.	Extracting SQL	No	N/A	N/A
Gould et al. [61]	They presented a static analysis technique for verifying the correctness of dynamically generated SQL query strings for database applications in Java.	Checking SQL	No	N/A	N/A
Annamaa et al. [20]	They described a tool that statically analyzes SQL queries embedded in Java programs.	Analyzing SQL	No	N/A	N/A
Lyu et al. [86]	They proposed a static analysis approach to detect SQL anti-patterns in mobile apps.	Detecting SQL anti-patterns	No	N/A	N/A
Our work	**We proposed an approach to locate the paths that lead to the generated SQL queries.**	**Locating paths for SQL**	**Yes**	**Yes**	**Yes**

(IRBL) aims to identify potentially buggy files by computing the similarity between a given software artifact (e.g., bug report) and source code files [149, 146, 81, 136, 143, 133]. The source code files are then ranked based on their similarity with the software artifact for investigation. Zhou et al. [158] proposed an IR-based method named *BugLocator* for locating relevant source code files based on initial bug reports by utilizing a revised Vector Space Model (rVSM) as well as similar bug information. Wong et al. [147] proposed an approach, *BRTracer*, which leverages two techniques segmentation and stack-trace analysis to improve the performance of bug localization. Wang and Lo [142] proposed an approach called *AmaLgam+* that integrates various information (e.g., version history, similar bug reports, and stack traces) to better locate buggy files given a bug report. Lee et al. [82] presented a comprehensive study that compares six state-of-the-art IR-based bug localization techniques. Pradel et al. [123] presented a technique *Scaffle* which uses crash reports to identify the possible file paths and the associated files that may have caused the crash. Chen et al. [35] proposed an IRBL approach, *Pathidea*, which leverages logs in bug reports to re-construct execution paths and they found that the execution path provides a significant improvement in bug localization accuracy. Other works on IRBL focus on optimizing and reformulating queries extracted from the bug report text [65, 64, 98, 55]. Similarly, our approach first applies static analysis to infer the database access (i.e., templated SQL query) for each control flow path. Then, we apply information retrieval (IR) techniques to find the control flow paths for which the inferred database accesses have the highest similarity with the given SQL query. Different from prior IRBL approaches that aim to locate bugs using bug reports, our approach is one of the first to apply IR techniques to locate the origin of SQL queries. Our approach also provides additional information (i.e., the code path) instead of only locating the method that generates the SQL query. Given a problematic SQL query, SLocator can locate the code path that generates the query.

4.3 Approach

As discussed in Section 4.2, there are various factors that affect how a SQL query is generated when using ORM. Hence, a simple text search based on the generated SQL query may not be sufficient to locate the origin (i.e., the control flow path, which contains a sequence of method calls) of the SQL query. Figure 5 provides an overview of our approach, SLocator, which automatically locates the origin of the SQL query. SLocator uses a combination of static analysis and information retrieval to locate the path. We first use static analysis to infer the database access of each control flow path in the source code. Then, given SQL queries, we use information retrieval techniques to rank the control flow paths that have the highest database access similarity (i.e., the similarity score between the database access inferred from the control flow path and the given SQL query).

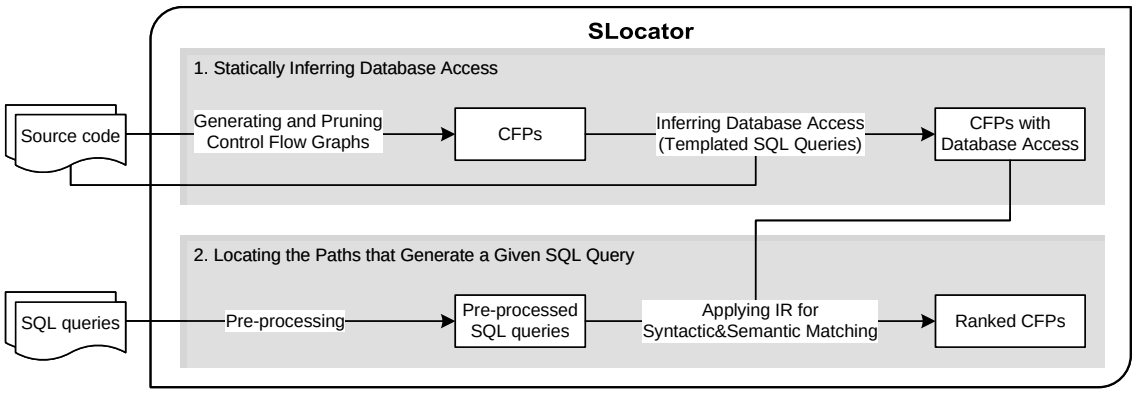


Figure 5: An overview of SLocator. CFP refers to control flow path and IR refers to information retrieval.

We implement SLocator in Java based on the Java Persistent API (JPA), which is Java's official specification for ORM frameworks. Below, we discuss the design of SLocator in detail.

4.3.1 Statically Inferring Database Access

4.3.1.1 Generating and Pruning Control Flow Graphs

To locate the possible control flow paths that generate a given SQL query, we use static analysis to construct the interprocedural control flow graph (CFG) of the application [131]. Specifically, the CFG is a directed graph, where the nodes represent the basic blocks and the edges connecting the nodes represent the transfer of control flow between basic blocks. We use Crystal[1], a Java static analysis framework that is built on top of Eclipse JDT, to analyze the source code and construct the CFG.

In database-backed web applications, users often interact with the applications by sending HTTP requests (e.g., using RESTFul APIs or through browsers) [14]. Therefore, SLocator statically analyzes the Java API for RESTful web services (JAX-RS) [109] specifications in the source code to identify a list of web request handling methods. An example of JAX-RS code is shown below:

```
@RequestMapping(value = "/owners/{ownerId}", method = RequestMethod.GET)
public Owner showOwner(int ownerId) {
    Owner owner = this.clinicService.findOwnerById(ownerId);
    return owner;
}
```

In this example, based on the JAX-RS annotations, when users send an HTTP GET request that ends with the URL "/owners/{ownerId}", method *showOwner* is called to handle the request.

[1] https://code.google.com/archive/p/crystalsaf/

The request handling methods are used as the entry points to the uncovered control flow graphs. For each request handling method, SLocator uncovers all of the associated control flow paths by traversing the interprocedural CFG. As the goal of SLocator is to statically locate the control flow path that results in generating a given SQL query, we omit cycles in the CFG. We perform a depth-first search (DFS) to traverse the CFG and omit the vertex that has been visited before (i.e., a cycle is detected). There may be multiple control flow paths that are associated with one request handling method, and not every path is related to database access. Hence, we further conduct pruning to remove the paths that do not have database access calls. In particular, we analyze if a path contains any API call to the EntityManager class (i.e., the main class in JPA for database accesses). We prune the path if it does not contain any call to EntityManager.

4.3.1.2 Statically Inferring the Database Access of Each Control Flow Path

When using ORM frameworks, many database accesses are abstracted as ORM API calls. As shown in Section 4.2, in most cases, developers only need to specify the association among classes (e.g., *OneToMany* or *OneToOne* relationships) and different database access configurations (e.g., *EAGER* or *LAZY*). Then, developers can access the DBMS by calling APIs such as EntityManager.find(User.class, userID). Therefore, to statically infer the database access (i.e., templated SQL queries), we analyze both the database access methods that are called on the control flow path and the corresponding ORM configuration. To infer the database access, we implement a database access translator that takes as input the tuple {database access method, entity mapping, association, retrieval strategy}, which are defined as:

- Database access methods: API calls to EntityManager.
- Entity Mapping: Annotations, such as *@Table* and *@Column*, which map an entity class to its corresponding database table.
- Associations: *ManyToOne*, *OneToMany*, *OneToOne*, and *ManyToMany*.
- Retrieval Strategy: *EAGER* or *LAZY*.

Table 7 shows the inferred database accesses given the database access APIs, entity mapping, association, and retrieval strategy. In addition to using method calls such as EntityManager.find(), JPA also provides native SQL to query database tables and JPA query language (JPQL) [78] to create queries against entities. Native SQL queries can be used in the method createNativeQuery(String queryString) while JPQL queries can be used in createQuery(String queryString), where queryString is the SQL query statement and JPQL query statement, respectively, to be executed. During our analysis, we analyze the abstract syntax tree (AST) of the program to extract the potential value of

the string variable (i.e., queryString) as the inferred database access (i.e., inferred queries). During the static analysis of the source code, we first use Eclipse JDT to create the AST for the method, which contains database access API calls. Then, we handle the argument of database access API calls, such as createQuery(String queryString). If the argument is a literal text, we extract it directly. If the argument is a variable, we try to extract its value in the AST based on the prior variable assignment. For the method createNamedQuery(String name), which supports both native SQL and JPQL, the inferred database access queryString is the corresponding named query (i.e., native SQL query or JPQL query) based on the name.

For direct calls to EntityManager APIs, the translator translates the Create, Read, Update, and Delete (CRUD) operations to the corresponding SQL queries using the SQL query templates shown in the table. For each operation, the translator inputs the parameters in the template, such as *table_name*, *column_name*, and *primary_key_name*, based on the entity mapping to generate the inferred SQL queries. When the association is *ManyToOne*, *OneToOne*, or when the retrieval strategy is *EAGER*, the inferred SQL queries for the EntityManager API calls would contain a *join* clause that selects data from two or more database tables [38, 78]. For JPQL, the inferred queries would contain multiple select statements to select the data records from the associated tables [12].

4.3.2 Locating the Paths that Generate a Given SQL Query

SLocator uses information retrieval techniques to locate the origin of a given SQL query. The SQL queries are used as the search term, and the corpus (i.e., collections of documents) is the inferred database access. Each document represents the inferred database access, with a mapping to the corresponding control flow path (as discussed in Section 4.3.1). SLocator compares both the syntactic and semantic similarity between the inferred database access and the given SQL query. SLocator returns a ranked list of the control flow paths whose inferred database accesses have the highest similarity with the SQL query. Below, we discuss the approach in detail.

4.3.2.1 Pre-processing SQL Queries

The SQL queries generated by the ORM frameworks may contain dynamic elements (e.g., aliases) that can affect localization accuracy. For example, SQL queries may contain dynamic values that cannot be found in the source code. Consider a SQL query from PetCinic that is generated by Hibernate:

```
select owner0_.id as id1_0_0_, ... pets1_.id as id1_1_1_, ... from owners owner0_ left
    outer join pets pets1_ on owner0_.id=pets1_.owner_id where owner0_.id=1
```

where *owner0_* and *pets1_* are aliases for the owners and pets tables, *id1_0_0_* and *id1_1_1_* are

Table 7: Translations from ORM API calls to inferred database accesses (templated SQL queries). For native SQL and JPQL, SQL statements or JPQL statements in queryString are extracted as inferred database accesses (inferred queries). Values in { } are statically inferred based on the entity mapping.

	Operation	Database Access API	Inferred Database Access	Inferred Database Access with EAGER retrieval
JPA Entity Manager	Create	persist(Object entity)	insert into {table_name} ({column_name}, ...) values(?, ...)	
	Read	find(Class entityClass, Object primaryKey)	select {column_name} ... from {table_name} where {primary_key_name}=?	select column_name ... from table_name [**join on {table_name.column_name} = {target_table_name.join_column_name}**] * where {primary_key_name}=?
	Update	merge(T entity)	update {table_name} set {column_name}=? ... where {primary_key_name}=?	
	Delete	remove(Object entity)	delete from {table_name} where {primary_key_name}=?	
Native SQL	CRUD	createNativeQuery(String queryString) createNamedQuery(String name)	queryString	
JPA Query Language	CRUD	createQuery(String queryString) createNamedQuery(String name)	queryString	queryString [**select {column_name} ... from {target_table_name} where {primary_key_name}=?**] *

aliases for the ID columns in the selected tables. Including such automatically-generated IDs and dynamic values/aliases will reduce the localization accuracy because they do not exist in the inferred database accesses (i.e., templated SQL queries). We pre-process the SQL queries by following pre-processing techniques that are used for software artifacts [81, 136, 143, 133, 43, 79, 24]. We first parse the SQL queries into abstract syntax trees (ASTs) and traverse the ASTs to remove automatically-generated variable and column names, and aliases. Then, we remove the dynamic values (i.e., string literals and numeric values). Finally, we transform all the words into lowercase. After the pre-processing steps, the above-mentioned SQL query becomes:

```
select from owners left outer join pets where owner.id=?
```

Once the SQL queries are pre-processed, we apply information retrieval to find the corresponding inferred database accesses that have the highest similarity.

4.3.2.2 Applying Information Retrieval for Syntactic and Semantic Matching

Given a SQL query (or a set of SQL queries), the goal is to find the inferred database accesses that have the highest similarity. In particular, SLocator compares both the syntactic and semantic similarity between the pre-processed SQL queries and the inferred database accesses.

Computing Syntactic Similarity. To compute the syntactic similarity, we represent both the pre-processed SQL query and the inferred database access as strings and calculate the similarity score [79, 73]. Given a SQL query q, SLocator computes the syntactic similarity as the cosine similarity between q and the inferred database access of a control flow path p as follows:

$$sim_{\text{syn}}(q, p) = cosine(\vec{q}, \vec{p}) = \frac{\vec{q} \cdot \vec{p}}{\|\vec{q}\| \cdot \|\vec{p}\|}, \tag{1}$$

where $\vec{q}$ and $\vec{p}$ are the weight vectors for the SQL query q and the inferred database access of a control flow path p, respectively. We compute the weight vectors based on the term frequency and inverse document frequency (i.e., $tf \cdot idf$), where more weights are given to words that have higher occurrences in a given document but have lower occurrences in the corpus (i.e., words that are more relevant).

Computing Semantic Similarity. As found in prior studies [79, 23], semantic information in SQL queries such as the accessed tables and operations on tables (e.g., select and update) are useful in identifying similar SQL queries. SLocator uses the Jaccard similarity index to compute the semantic similarity between a SQL query q and the inferred database access of a control flow path p as follows:

$$sim_{\text{sem}}(q, p) = \frac{|features(q) \cap features(p)|}{|features(q) \cup features(p)|}, \tag{2}$$

where *features(p)* is the set of accessed tables and CRUD operations on tables in p. Intuitively, if the accessed tables and operations are different between p and q, it is less likely that the two database accesses are similar.

Combining Similarity Scores and Deriving Path Ranking. We combine the semantic and syntactic similarity to measure the similarity score between a SQL query q and an inferred database access of a path p as follows:

$$Score(q, p) = sim_{\text{syn}}(q, p) + sim_{\text{sem}}(q, p) \tag{3}$$

$Score(q, p)$ ranges between 0 and 2, where the larger the value the higher the similarity. Given q, we compute the $Score(q, p_i)$ for every control flow path p_i generated by the previous steps in our approach. The p_i with a higher similarity score would be ranked higher in the result and is more likely to be the path that generates q.

4.4 Evaluation

In this section, we first introduce the studied applications and experimental setup. Then, we evaluate SLocator by answering three research questions (RQs). For each RQ, we discuss the motivation, approach, and results.

4.4.1 Evaluation Setup

Studied Applications. We conduct our study on seven open source applications that are popular (i.e., with an average of 1.4K stars on GitHub), have a long development history, or have been used in prior studies on database-backed applications [38, 39, 130, 27, 48, 32, 145]. Table 8 shows an overview of the studied applications, such as the number of commits, database tables, Java files, and distinct database accesses. On average, there are 33 database tables, 463 Java source code files, and 85 distinct database accesses where the SQL queries may be generated. The database-backed web applications are implemented in Java using JPA to access the database. Among all the 595 database accesses, 113 (19.0%) use JPA API persist(), 59 (9.9%) use JPA API find(), 54 (9.1%) use JPA API merge(), 67 (11.3%) use JPA API remove(), 291 (48.9%) use JPQL queries, and 11 (1.8%) use JPA criteria (the statistics of JPA API usage can be found in the online appendix [11]). No native SQL queries are used in the studied applications. Hence, given a large number of database access calls, manual analysis of the origin of a SQL query can be difficult. PetClinic [120] is developed and maintained by Pivotal Software for showcasing standard practices in developing database-backed web applications. CloudStore [45] is an e-commerce web application that is developed according to the TPC-W benchmark [137] while BroadleafCommerce [29] is an enterprise e-commerce framework.

Table 8: An overview of the studied applications. DB access refers to database access.

Application	Version	LOC	No. of commits	No. of tables	No. of Java files	No. of distinct DB accesses
PetClinic	1.5	2.4K	707	7	38	12
CloudStore	2.0	11.2K	200	11	98	40
WallRide	1.0.0.M18	32.6K	744	35	363	93
JeeWeb	1.0	40.8K	64	31	419	112
PublicCMS	4.0	47.3K	1,103	43	496	132
bbs	5.6	129K	40	44	579	148
BroadleafCommerce	6.0.11-GA	197K	17,599	60	1,284	58
Avg. across applications	–	65.8K	2,922	33	463	85

Since Broadleaf is a framework, we study the site module provided by BroadleafCommerce, which uses Broadleaf's APIs to build an online shopping website. PublicCMS [124] and WallRide [141] are content management systems (CMSs). JeeWeb [74] is a development system that helps developers generate source code. bbs [25] is a forum application and we study the admin module that is used to manage the forum. In particular, PublicCMS is developed/maintained by a company, has over 1.6K stars on GitHub, is used in many commercial settings, and has many users around the world. BroadleafCommerce has been developed since 2009 and has over 1.5K stars on GitHub.

Experimental Setup. We deploy the studied applications on Tomcat 7, using MySQL 5.6 as the database management system. To simulate a real-world deployment setting, we follow a prior study and populate the main database tables to 20,000 records [151]. For the applications that already contain initial data records, we duplicate these records while keeping their association relationships. For the applications that do not have initial data records, we exercise them by simulating user actions to generate data records and populate the databases. To evaluate SLocator, we exercise the applications by running simulated workloads after the data is populated, and record the application execution information. We first analyze the application usage and then use JMeter [56] to automatically send user requests to simulate hundreds of concurrent users. For each user, we set JMeter to generate random values for variables in the request. Hence, given hundreds of concurrent users, each request would be called hundreds of times with random input values.

The workload covers most of the application web pages by navigating the menu. For PetClinic, the workload covers user actions such as searching and adding/modifying owners' and pets' information. For CloudStore and BroadleafCommerce, the workload covers browsing, searching for items,

Table 9: Statistics of running SLocator against the studied applications. Time to locate the paths refers to the average time to rank and locate the control flow paths for a given SQL query.

Application	No. of inferred CFPs	Static analysis execution time (s)	Time to locate the paths (ms)
PetClinic	18	13	7
CloudStore	64	48	12
WallRide	487	175	142
JeeWeb	333	102	20
PublicCMS	1,267	318	41
bbs	2,298	162	120
BroadleafCommerce	1,317	371	679
Avg. across applications	826	170	146

adding items to carts, and checking out. For WallRide and PublicCMS, the workload covers common actions in CMS such as editing user profiles, adding content (e.g., pictures), editing/adding posts, and editing web pages. For JeeWeb, the workload covers editing/adding system content (e.g., user, department, role), configuring the database, and generating source code to query the database tables. For bbs, the workload covers common actions in forums such as writing posts and questions, editing/adding tags for posts and questions, and viewing posts and questions. Overall, the workload covers 71.2% of the related web requests, 70.7% of the related database accesses, and 74% of the related database tables. Both the workload and SQL queries generated by the workload are publicly available (the statistics of the workload and SQL queries can be found in the online appendix) [11].

Statistics of SLocator. Table 9 shows the statistics of running SLocator against the studied applications. On average, there are 826 control flow paths that contain database access calls leading to generating SQL queries. Note that, each database access may generate multiple SQL queries based on the ORM configuration and each control flow path may contain several database accesses. Hence, the number of generated SQL queries would be even larger, which makes manual analysis of the origin of a SQL query more difficult. We conduct all of our experiments on a Windows 10 machine with an Intel Core i5 CPU@1.70GHz and 16GB of RAM. On average, SLocator takes 170 seconds to statically analyze the source code to infer control flow paths with database access details. SLocator takes an average of 146 milliseconds to rank and locate the control flow paths for given SQL queries. For each release of the application, the static analysis only needs to be executed once. Thus, the performance overhead of SLocator is relatively small.

Approaches and Metrics for Evaluating SLocator. In regular usage of SLocator, we would not need any instrumentation. However, to evaluate the localization accuracy of SLocator, we use AspectJ [57] to instrument the application to get the ground truth (i.e., the web request handling methods and the control flow paths that generate the given SQL query). We only apply instrumentation to get the ground truth. First, we set the configuration of AspectJ to define the pointcuts to match all the methods within the application source code. During the execution of the workloads, for each user request, AspectJ records all the methods that are executed and the corresponding SQL queries that are sent to the DBMS (i.e., MySQL), which represents the dynamic execution path (i.e., the ground truth). Then, given a SQL query, we apply SLocator to find its origin and compare the origin with the ground truth in the evaluation step. For replication purposes, we make our AspectJ configuration publicly available [11].

We define that a dynamic execution path, d, matches with the statically uncovered control flow path, p, if $p \subset d$. Namely, if every method in p appears in d in the same order, we say that p matches with d (i.e., an ordered set). We define the matching using a subset due to two reasons. First, there may be calls to external frameworks in the dynamic execution paths, which may not be captured in the statically uncovered control flow paths. Second, there may be repeated method calls in the dynamic execution path.

Below, we define the information retrieval metrics that we use to evaluate the effectiveness of SLocator when locating the paths that generate the SQL queries.

Top@K. This metric calculates the percentage of the SQL queries whose dynamic execution path matches with one of the top K results, i.e., successfully located.

Precision@K. Given a SQL query, this metric calculates the percentage of the paths that are correctly located within the given top K results.

$$Precision(K) = \frac{\#\text{ correctly located paths in top } K}{K} \tag{4}$$

Mean Average Precision (MAP). Given a SQL query, this metric first calculates the average precision (AP) for every path in the ranked paths as follows:

$$AP = \sum_{i=1}^{N} \frac{Precision(i) \times pos(i)}{\text{total } \#\text{ of correctly located paths}} \tag{5}$$

where N is the number of ranked paths and $pos(i)$ is an indicator function. $pos(i) = 1$ if the i^{th} path correctly matches with the dynamic execution path. Otherwise, $pos(i) = 0$. For computing MAP, we take the average AP of all the given SQL queries.

Mean Reciprocal Rank (MRR). The reciprocal rank for a SQL query is the reciprocal of the position of the first correctly matched path in the ranked results. This metric calculates the mean

of the reciprocal ranks across all SQL queries:

$$MRR = \frac{1}{M}\sum_{j=1}^{M}\frac{1}{rank_j} \tag{6}$$

where M is the number of given SQL queries and $rank_j$ means the position of the first correctly matched path in the ranked list for the j^{th} SQL query.

4.4.2 RQ1: How effectively can SLocator locate the code path that generates a given SQL query?

Motivation. Due to the discrepancy between the application code and the generated SQL queries, locating where a given SQL query is generated can be a challenging task. In this RQ, we evaluate how well SLocator can locate the paths that generated a given list of SQL queries.

Approach. In production settings, developers often only have access to DBMS logs, where DBMS (e.g., MySQL) records the SQL queries that it executes. DBMS logs often record possibly problematic SQL queries for diagnosing database access issues (e.g., slow SQL queries or SQL queries that caused database deadlocks) [114, 121]. We retrieve DBMS logs from MySQL and use such logs as the input to SLocator to evaluate its effectiveness. Note that, as described in Section 4.4.1, we obtain the dynamic execution paths that generate the SQL queries (i.e., the ground truth) by instrumenting the applications using AspectJ. For every SQL query recorded in the DBMS log, we map it to the corresponding SQL query and dynamic execution path captured by AspectJ.

We evaluate SLocator by using two types of DBMS logs: individual query log and SQL session log. In the individual query log, MySQL records the execution of individual SQL queries. In the SQL session log, MySQL records all the SQL queries that it executes and groups the queries based on sessions (i.e., connections). Listing 4.1 shows an example of SQL session log from General Query Log [115] in MySQL which has three columns: session ID, command, and argument.

Listing 4.1: An example of SQL session log.

```
1 8 Query set session transaction read only
2 8 Query SET autocommit=0
3 8 Query select owner0_.id as id1_0_0_, ... from owners owner0_ left outer join pets pets1_ on
      owner0_.id=pets1_.owner_id where owner0_.id=1
4 8 Query select pettype0_.id as id1_3_0_, ... from types pettype0_ where pettype0_.id=1
5 8 Query select visits0_.pet_id as pet_id4_1_0_, visits0_.id as id1_6_0_, ... from visits
      visits0_ where visits0_.pet_id=1
6 8 Query commit
```

For instance, 8 is the session ID, "Query" is the command, and the rest are arguments (i.e., actual SQL query). The session starts with *set session transaction read only*, *SET autocommit=0* (Lines 1-2) and ends with *commit* (Line 6). We identify the session based on ID and extract query statements on Lines 3-5 as input for SLocator. Since the SQL queries are executed in the same connection, they reflect that the queries are generated by one sequential execution in the application (i.e., from the same execution path).

We report two levels of granularity in the SQL query localization results: web request and control flow path. In database-backed web applications, most user actions are handled by various web requests. Therefore, identifying the correct web request and the corresponding request handling methods (i.e., the root of the control flow path) that generate a given SQL query provides an important starting point for investigation. We also report the localization results at a finer-grained level, namely, whether SLocator can locate the execution path that generates a given SQL query.

We also compare SLocator with a baseline approach, which applies text search to locate the origin of a given SQL query at the level of web request handling method (the baseline approach does not contain the static analysis component so it cannot locate control flow paths). Given a SQL query, we build a corresponding query template and search for matching database accesses in the source code. We use query templates instead of directly using the given SQL query because there may be major differences between the generated SQL queries and the database accesses (JPQL or calls to EntityManager) in the source code [78] (e.g., generated SQL queries may have aliases as discussed in Section 4.2). The query template consists of keywords such as the CRUD operations (e.g., SELECT, UPDATE, INSERT, and DELETE) and the database tables and conditions used in the given SQL query. For calls to ORM APIs (i.e., EntityManager), the query template consists of keywords such as the ORM API calls and entity names inferred from the given SQL query. For example, given a SQL query SELECT * from owners, we infer the ORM API call EntityManager.find() and the entity name Owner. We rank the matching database accesses by calculating the cosine similarity between the query template and the database accesses. Finally, for each matched database access call, we analyze its call hierarchy to find the corresponding web request handling method as the origin of the SQL query.

Results. Table 10 shows the localization results of using SQL session logs. Overall, we find that SLocator has a high Top@K when locating both the web request and the control flow path that generates the SQL queries. When $K = 1$ and $K = 5$, SLocator achieves an average Top@K of 54.0% and 88.8% when locating control flow paths, and 60.2% and 91.7% when locating web requests. The average MAP and MRR are 0.60 and 0.72 when locating control flow paths, and 0.63 and 0.75 when locating web requests. Table 11 shows the localization results of using individual query logs. Compared to the localization results of using SQL session logs, the localization performance is lower

when using individual query logs. The average Top@K is 48.3% for control flow paths and 54.4% for web requests when $K = 1$. When $K = 5$, Top@K is 74.7% and 79.6% for control flow paths and web requests, respectively. Correspondingly, the average MAP and MRR are 0.47 and 0.64 when locating control flow paths, and 0.53 and 0.68 when locating web requests.

SLocator has a much better localization result compared to the baseline. The baseline approach achieves an average Top@5, MAP, and MRR of 28.2%, 0.21, and 0.22, respectively, when using SQL session logs. When using individual query logs, the baseline achieves an average Top@5, MAP, and MRR of 18.4%, 0.15, and 0.15, respectively. Compared to the baseline, SLocator improves the Top@5, MAP, and MRR by 225%, 200%, and 241%, respectively, when using SQL session logs. When using individual query logs, the improvement of Top@5, MAP, and MRR is by 333%, 253%, and 353%, respectively.

We find that the decrease in Top@K when using individual query logs is because there may be multiple control flow paths or web requests that can generate the same SQL query. For example, in PetClinic, the application generates a SQL query to select pets' visit information: select from visit where pet_id=1, which may come from six different request handling methods: processFindform, initCreationForm, showOwner, initUpdateOwnerForm, processUpdateOwnerForm, and processUpdate-Form. Even for one request handling method showOwner, this SQL query may come from three different paths. In total, the SQL query may come from nine control flow paths. Therefore, the localization accuracy decreases when using individual query logs.

In contrast, there may be fewer web requests and control flow paths that generate a given SQL session log. SQL queries in the SQL session log are more likely generated by the same business logic (e.g., the same web request). Hence, it is less likely that multiple web requests and control flow paths generate the same set of SQL queries. For example, in PetClinic, the SQL session log shown in the previous example in Listing 4.1 may come from only two request handling methods (i.e., showOwner and initUpdateOwnerForm) and four control flow paths. However, if we consider the individual SQL query on Line 5, it may be generated by six request handling methods and nine control flow paths. Nevertheless, our findings show that SLocator can still achieve good accuracy when localizing the path given one single SQL query.

We apply SLocator on the seven studied applications and find that, on average, developers need to investigate two control flow paths (the paths have an average of six methods) when using SQL session logs, and five control flow paths (the paths have an average of eight methods) when using individual query logs, to find the origin of the SQL query. Hence, developers do not need to investigate many returned paths to find the correct SQL origin when using SLocator.

Discussion. As shown in Tables 10 and 11, SLocator has a very high Top@K across the studied applications. However, we find that even if we increase K (e.g., set K to 10 or 20), the numbers

Table 10: The localization results when using SQL session logs. Request-Baseline refers to locating the web request using the baseline approach. Request-SLocator and Path-SLocator refer to using SLocator to locate the web request and control flow path, respectively.

Application	Matching	Top@K			MAP	MRR
		K=1	K=3	K=5		
PetClinic	Request-Baseline	9.5%	14.3%	14.3%	0.12	0.12
	Request-SLocator	52.4%	95.2%	95.2%	0.67	0.76
	Path-SLocator	52.4%	95.2%	95.2%	0.67	0.76
CloudStore	Request-Baseline	28.6%	42.9%	42.9%	0.31	0.35
	Request-SLocator	71.4%	85.7%	92.9%	0.79	0.80
	Path-SLocator	57.1%	85.7%	92.9%	0.71	0.73
WallRide	Request-Baseline	11.4%	22.7%	27.3%	0.17	0.18
	Request-SLocator	59.1%	79.5%	95.5%	0.62	0.73
	Path-SLocator	54.5%	77.3%	95.5%	0.65	0.71
JeeWeb	Request-Baseline	4.0%	16.2%	16.2%	0.09	0.09
	Request-SLocator	40.4%	85.9%	90.9%	0.38	0.65
	Path-SLocator	37.4%	78.8%	82.8%	0.35	0.66
PublicCMS	Request-Baseline	10.0%	20.0%	26.0%	0.16	0.17
	Request-SLocator	86.0%	96.0%	98.0%	0.75	0.93
	Path-SLocator	70.0%	88.0%	94.0%	0.67	0.83
bbs	Request-Baseline	31.4%	34.3%	35.7%	0.32	0.33
	Request-SLocator	64.3%	85.7%	91.4%	0.62	0.77
	Path-SLocator	58.6%	78.6%	82.9%	0.57	0.73
Broadleaf-Commerce	Request-Baseline	26.1%	34.8%	34.8%	0.30	0.30
	Request-SLocator	47.8%	60.9%	78.3%	0.59	0.60
	Path-SLocator	47.8%	60.9%	78.3%	0.56	0.60
Avg. across applications	Request-Baseline	17.3%	26.5%	28.2%	0.21	0.22
	Request-SLocator	60.2%	84.1%	91.7%	0.63	0.75
	Path-SLocator	54.0%	80.6%	88.8%	0.60	0.72

Table 11: The localization results when using individual query logs. Request-Baseline refers to locating the web request using the baseline approach. Request-SLocator and Path-SLocator refer to using SLocator to locate the web request and control flow path, respectively.

Application	Matching	Top@K			MAP	MRR
		K=1	K=3	K=5		
PetClinic	Request-Baseline	15.4%	23.1%	23.1%	0.19	0.19
	Request-SLocator	69.2%	92.3%	100.0%	0.65	0.82
	Path-SLocator	69.2%	92.3%	100.0%	0.65	0.82
CloudStore	Request-Baseline	16.1%	19.4%	19.4%	0.18	0.18
	Request-SLocator	61.3%	77.4%	87.1%	0.61	0.71
	Path-SLocator	45.2%	61.3%	80.6%	0.47	0.58
WallRide	Request-Baseline	8.0%	13.6%	15.9%	0.11	0.11
	Request-SLocator	35.2%	54.5%	60.2%	0.36	0.50
	Path-SLocator	30.7%	52.3%	59.1%	0.34	0.47
JeeWeb	Request-Baseline	3.3%	10.6%	10.6%	0.06	0.06
	Request-SLocator	52.0%	87.0%	90.2%	0.46	0.72
	Path-SLocator	48.8%	78.0%	80.5%	0.42	0.74
PublicCMS	Request-Baseline	10.8%	18.9%	18.9%	0.14	0.14
	Request-SLocator	62.2%	71.6%	73.0%	0.55	0.74
	Path-SLocator	52.7%	66.2%	68.9%	0.47	0.66
bbs	Request-Baseline	26.0%	30.0%	30.0%	0.28	0.28
	Request-SLocator	53.0%	69.0%	77.0%	0.52	0.68
	Path-SLocator	48.0%	63.0%	71.0%	0.48	0.68
Broadleaf-Commerce	Request-Baseline	8.7%	10.9%	10.9%	0.10	0.10
	Request-SLocator	47.8%	63.0%	69.6%	0.54	0.59
	Path-SLocator	43.5%	54.3%	63.0%	0.44	0.53
Avg. across applications	Request-Baseline	12.6%	18.1%	18.4%	0.15	0.15
	Request-SLocator	54.4%	73.5%	79.6%	0.53	0.68
	Path-SLocator	48.3%	66.8%	74.7%	0.47	0.64

still may not reach 100%. The finding shows that there may be some SQL queries for which we cannot find the corresponding statically inferred control flow path. After some manual investigation, we find that such mismatches are caused by the limitation of static analysis and the frameworks that these applications use. For example, PetClinic uses the Spring framework [13] for web request handling and adds the *@ModelAttribute*="visit" annotation to the method loadPetWithVisit(). The method loadPetWithVisit() contains a database access call, but the method is not used in all the Java files. We find that the model attribute (i.e., "visit") is referenced in one of the JSP (Java Server Page) files that takes input from the user. In other words, loadPetWithVisit() is called automatically when a user submits a web form to the application server, which is the reason why SLocator was not able to find the control flow path that generates the given SQL query. The issue is common across the studied applications. As another example, in WallRide, developers override the method postHandle() from the Spring framework. postHandle() is executed automatically after handling each web request, and the overridden postHandle() contains database access calls.

In short, even though our results show that SLocator is able to locate the path where a given SQL query is generated with good accuracy, there are still some limitations caused by static analysis. Future studies may consider the frameworks that the application uses to increase the accuracy of static analysis.

We find that SLocator achieves good localization accuracy. When using SQL session logs, the origin (i.e., the CFP) of 54% of the SQL queries can be located in Top@1, and almost 89% can be located in Top@5. When using individual SQL queries, the origin of more than 48% of the SQL queries can be located in Top@1, and almost 75% can be located in Top@5. On average, developers need to investigate two and five control flow paths to find the origin when using SQL session logs and individual SQL queries, respectively.

4.4.3 RQ2: What is the localization accuracy for SQL queries with different lengths?

Motivation. In RQ1, we evaluate the overall localization accuracy of SLocator. However, the recorded SQL queries may have different complexities such as lengths which may affect how SLocator performs. In this RQ, we evaluate the accuracy of SLocator in localizing the paths for SQL queries with different lengths (i.e., the number of words involved).

Approach. The goal of RQ2 is to assess the ability of SLocator to locate SQL queries with different lengths. Since the range of SQL query lengths varies in the studied applications, instead of using a pre-defined threshold for all the applications, we classify the length of the SQL queries in each

Table 12: The localization results for SQL queries with different lengths (i.e., bottom, middle, and top) when using individual query logs. The length of SQL queries is measured using the number of words and is classified into three buckets based on the quantiles (i.e., bottom 1/3, middle 1/3, and top 1/3). Request and Path refer to using SLocator to locate the web request and control flow path, respectively. SQL lengths refer to the range of SQL query lengths.

Application	Matching	Bottom length						Middle length						Top length					
		SQL lengths	Top@K			MAP	MRR	SQL lengths	Top@K			MAP	MRR	SQL lengths	Top@K			MAP	MRR
			K=1	K=3	K=5				K=1	K=3	K=5				K=1	K=3	K=5		
PetClinic	Request	7–9	75.0%	100.0%	100.0%	0.58	0.88	10–31	75.0%	100.0%	100.0%	0.71	0.88	59–113	60.0%	80.0%	100.0%	0.66	0.74
	Path		75.0%	100.0%	100.0%	0.58	0.88		75.0%	100.0%	100.0%	0.71	0.88		60.0%	80.0%	100.0%	0.66	0.74
CloudStore	Request	5–20	80.0%	90.0%	90.0%	0.69	0.84	21–68	60.0%	80.0%	90.0%	0.60	0.73	79–266	45.5%	63.6%	81.8%	0.55	0.58
	Path		60.0%	60.0%	90.0%	0.44	0.67		40.0%	70.0%	80.0%	0.50	0.57		36.4%	54.5%	72.7%	0.48	0.50
WallRide	Request	5–12	51.7%	72.4%	86.2%	0.51	0.64	12–57	24.1%	44.8%	44.8%	0.25	0.34	57–990	30.0%	46.7%	50.0%	0.35	0.42
	Path		44.8%	69.0%	82.8%	0.48	0.60		20.7%	44.8%	44.8%	0.24	0.32		26.7%	43.3%	50.0%	0.33	0.40
JeeWeb	Request	5–10	65.9%	95.1%	95.1%	0.52	0.81	10–51	26.8%	85.4%	90.2%	0.36	0.57	51–141	63.4%	80.5%	85.4%	0.50	0.74
	Path		63.4%	87.8%	87.8%	0.49	0.76		24.4%	75.6%	78.0%	0.31	0.50		58.5%	70.7%	75.6%	0.46	0.67
PublicCMS	Request	5–6	75.0%	83.3%	87.5%	0.70	0.80	6–22	64.0%	72.0%	72.0%	0.54	0.69	23–106	48.0%	64.0%	64.0%	0.52	0.56
	Path		70.8%	83.3%	87.5%	0.60	0.78		56.0%	68.0%	68.0%	0.53	0.63		48.0%	64.0%	64.0%	0.46	0.56
bbs	Request	5–7	75.8%	84.8%	93.9%	0.71	0.82	7–21	69.7%	81.8%	81.8%	0.57	0.75	22–100	14.7%	41.2%	55.9%	0.28	0.33
	Path		69.7%	78.8%	87.9%	0.65	0.76		60.6%	69.7%	69.7%	0.53	0.65		14.7%	41.2%	55.9%	0.27	0.33
Broadleaf-Commerce	Request	5–11	33.3%	46.7%	66.7%	0.49	0.45	13–39	40.0%	46.7%	46.7%	0.46	0.47	39–447	37.5%	62.5%	75.0%	0.46	0.52
	Path		33.3%	46.7%	66.7%	0.46	0.45		40.0%	40.0%	46.7%	0.36	0.45		37.5%	62.5%	75.0%	0.45	0.52
Avg. across applications	Request	–	65.2%	81.8%	88.5%	0.60	0.75	–	51.4%	73.0%	75.1%	0.50	0.63	–	42.7%	62.6%	73.2%	0.47	0.56
	Path	–	59.6%	75.1%	86.1%	0.53	0.70	–	45.2%	66.9%	69.6%	0.45	0.57	–	40.3%	59.5%	70.5%	0.44	0.53

studied application into three buckets based on the quantiles (i.e., bottom 1/3, middle 1/3, and top 1/3). We use length (number of words) as a proxy for the complexity of a SQL query, whereas a longer SQL query has a higher complexity. We evaluate the effectiveness of SLocator on localizing the paths for SQL queries in each studied application across the three length groups.

Results. Table 12 shows the localization results for SQL queries at different length groups (i.e., bottom 1/3, middle 1/3, and top 1/3) when using individual query logs. We observe that, in most studied applications, bottom-length SQL queries get better localization results than middle-length SQL queries, which in turn get better localization results than top-length SQL queries. When locating web requests for SQL queries that belong to the three length groups, the average Top@1 are 65.2%, 51.4%, and 42.7% while the Top@5 are 88.5%, 75.1%, and 73.2%, respectively. When locating control flow paths for SQL queries that belong to the three length groups, the average Top@1 are 59.6%, 45.2%, and 40.3% while the Top@5 are 86.1%, 69.6%, and 70.5%, respectively. We find that the decrease in Top@K for SQL queries at different length groups is because longer SQL queries have more words involved, and therefore, are harder to be matched with corresponding control flow paths' inferred database accesses compared to shorter SQL queries. Nevertheless, SLocator achieves good localization results for SQL queries with different lengths. For the SQL queries with length in the top 1/3, the average Top@5 are 73.2% and 70.5% for web requests and control flow paths,

respectively. Compared to the result in RQ1, the decreases are by 8% and 6%, respectively. These results suggest that SLocator can be used to effectively locate the code path for SQL queries at different length groups.

We find that SLocator achieves better localization results for short SQL queries compared to long SQL queries. For the SQL queries with length in the top 1/3, SLocator achieves good localization results - the average Top@5 are 73.2% and 70.5% for web requests and control flow paths, respectively.

4.4.4 RQ3: Can SLocator help localize issues in database-backed web applications?

Motivation. Database access performance is critical in database-backed applications since it directly affects the user-perceived quality [151, 38, 150]. Most DBMSs record slow SQL queries and database deadlocks for developers to conduct further investigation [116, 113, 112]. Slow SQL log records the SQL queries that take longer than a predefined threshold (e.g., one second) to execute. Such slow SQL queries may indicate performance issues or opportunities for performance optimization. Deadlock log records the SQL queries that were blocked when deadlocks happen. In database-backed applications, each database transaction may execute multiple SQL queries. Deadlocks happen when two or more database transactions are waiting for one another to release locks. Database deadlock is one of the main reasons for major performance degradation [40, 62].

In the previous RQs, we evaluate the overall localization accuracy of SLocator. In this RQ, we conduct two case studies, i.e., slow SQL queries and SQL queries that cause database deadlocks, to illustrate how SLocator helps localize database access issues in database-backed web applications.

Approach. As described in Section 4.4.1, we populate the data in the database since many performance issues only occur under large loads [38, 150]. We evaluate SLocator using either existing slow SQL queries or injected database deadlocks. Below, we discuss how we trigger/inject the performance issues.

Triggering Slow Queries: To trigger slow queries, we exercise the applications by running the same workload that we used in RQ1 and configure MySQL to record the execution time of each SQL query. Then, we calculate the average execution time for each unique SQL query and take the top 10% most time-consuming queries as slow SQL queries by following a prior study [151].

Injecting and Triggering Database Deadlocks: Injecting deadlocks requires much manual effort, and a deep understanding of the database tables and the business logic of the system. Therefore, we choose WallRide, a medium size application with 35 database tables, to inject a deadlock. The size

of WallRide is not too small for a study on deadlocks and is feasible for manual study the application source code to inject a deadlock. However, since SLocator achieves similar localization results across the studied applications, we believe SLocator can still help locate the origin of deadlocking SQL queries in other applications. We inject a deadlock in WallRide by changing the lock model type from *PESSIMISTIC_ WRITE* (i.e., pessimistic write lock) to *NONE* (i.e., no lock) [110] (Lines 3-4), as shown in Listing 4.2.

The method *PostRepositoryImpl.lock* is called before retrieving data from the DBMS and locks the corresponding database records with a pessimistic write lock. A pessimistic write lock is an exclusive lock in MySQL [97], which prevents concurrent writing of the same records and reduces the likelihood of deadlock in the database. By changing the lock to *NONE*, there will be chances that a deadlock may happen. After injecting the deadlock, we build and deploy the modified application. We use JMeter to automatically send user requests and simulate hundreds of concurrent users to trigger the deadlock.

Listing 4.2: Database deadlock injected in WallRide.

```
1 public void PostRepositoryImpl.lock(long id) {
2 ...
3 - entityManager.createQuery(query).setLockMode(LockModeType.PESSIMISTIC_WRITE).getSingleResult();
4 + entityManager.createQuery(query).setLockMode(LockModeType.NONE).getSingleResult();
```

Results.

Slow Queries: We give an example from PetClinic to demonstrate how SLocator locates the origin of a slow SQL query. The slow SQL log identifies the following SQL query as slow in PetClinic:

```
select distinct owner0_.id as id1_0_0_, ... from owners owner0_ left outer join pets
    pets1_ on owner0_.id=pets1_.owner_id where owner0_.last_name like '%'
```

The SQL query searches for the owner whose last name matches any string (i.e., like '%', where the wildcard '%' means a string with zero or more characters). By using this SQL query as the input to SLocator, SLocator returns the control flow path as shown in Listing 4.3 as the first ranked result. To gain more information about the inferred control flow paths, SLocator also returns calls to third-party libraries in the returned path. For instance, the request method processFindForm calls the methods findOwnerByLastName and java.util.Map.put, while the generic method java.util.Map.put is from Java's util library. The method findByLastName accesses the DBMS and generates three SQL queries Q1, Q2, and Q3, where the slow query is generated (i.e., Q1).

Listing 4.4 shows the corresponding source code containing the origin of the slow SQL query.

Based on the control flow path returned by SLocator, the potential execution path of the source code covers Line 3 and Lines 9-10 (as highlighted in blue). Line 10 indicates that all the owners retrieved from the DBMS will be displayed on one web page ownersList.html. The performance issue occurs when there are many owners whose last name matches the given search string. For this particular SQL query, it retrieves and displays all the users. A solution is to add pagination so that only a limited number of owners would be retrieved and displayed for every page. Note that, in this example, the performance issue would not occur if the code executes the first or the second branch (i.e., the number of matched owners is zero or one). Hence, the control flow path that is returned by SLocator may provide additional information to locate performance issues.

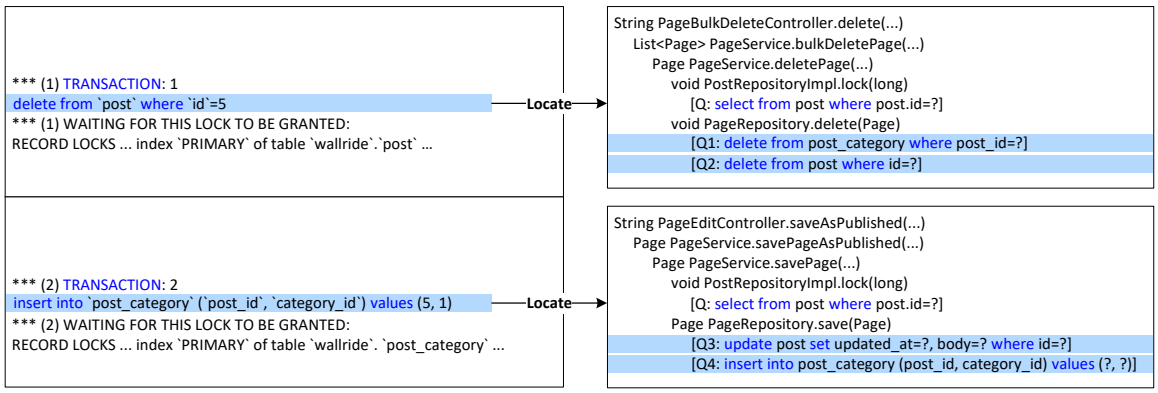
Listing 4.3: The control flow path and inferred database access returned by SLocator for a slow SQL query in PetClinic.

```
1 String ownercontroller.processFindForm(Owner, BindingResult, Map)
2     Collection<Owner> ClinicServiceImpl.findOwnerByLastName(String)
3         Collection<Owner> JPAOwnerRepositoryImpl.findByLastName(String)
4             [Q1: select distinct owner from owner owner left join fetch owner.pets where
                  owner.lastname like :lastname]
5             [Q2: select id, name from types where id=?]
6             [Q3: select id, visit_date, description from visits where id=?]
7     V java.util.Map.put(K, V)
```

Listing 4.4: Source code containing the origin of the slow SQL query.

```
1  String ownercontroller.processFindForm(Owner, BindingResult, Map){
2      // find owners by last name
3      Collection<Owner> results = this.clinicService.findOwnerByLastName(owner.getLastName());
4      if (results.isEmpty()) {                // branch 1: no owners found
5          ...
6      } else if (results.size() == 1) {       // branch 2: 1 owner found
7          ...
8      } else {                                // branch 3: multiple owners found
9          model.put("selections", results);
10         return "owners/ownersList";
11     }
12 }
```

Database Deadlocks: We use the injected deadlock in WallRide to illustrate the usage of SLocator to locate the origin of deadlock SQL queries for further diagnosis. Figure 6a shows the deadlock log obtained by using the MySQL command SHOW ENGINE INNODB STATUS. Two SQL queries are



(a) Deadlock log obtained from MySQL. A deadlock happens when two transactions T1 and T2 access database tables post and post_category.

(b) The control flow paths and inferred database access for the deadlock SQL queries.

Figure 6: Using SLocator to locate the paths in the control flow graphs that result in generating deadlock SQL queries.

blocked (as highlighted in blue), waiting for a lock to be granted in transactions T1 (TRANSACTION: 1) and T2 (TRANSACTION: 2), respectively. However, it is unknown how this deadlock happens according to the log because these two SQL queries should not block each other as they access different tables (i.e., post and post_category).

By using the first blocked SQL query in transaction T1 as the input to SLocator, SLocator returns the first control flow path shown in Figure 6b. Note that the path returned by SLocator is one specific path in the control flow graph, so the methods are on the same call path (i.e., no branching in between). The request handling method PageBulkDeleteController.delete (i.e., the root of the returned control flow path) calls the method PageService.bulkDeletePage, which in turn calls the method PageService.deletePage. PostRepositoryImpl.lock accesses the DBMS and generates the SQL queries Q. PageRepository.delete accesses the DBMS and generates two SQL queries Q1 and Q2, where SLocator locates Q2 as where the first blocked SQL query is generated. Note that Q and Q1 must have been executed since Q, Q1, and Q2 are on the same control flow path. Similarly, using the second blocked SQL query in Figure 6a as the input, SLocator returns the second control flow path shown in Figure 6b.

Based on the control flow paths that are returned by SLocator, we can see that Q1 and Q4 access the post_category table, and Q2 and Q3 access the post table. Since Q1 and Q4, and Q2 and Q3 are in different methods, a deadlock may happen when the two methods are executed by two separate transactions. For example, a transaction T1 may hold the lock on the post_category table (i.e., executing Q1) while another transaction T2 holds the lock on the post table (i.e., executing Q3). In this case, T1 cannot execute Q2 because T2 is holding the lock; and T2 cannot execute Q4 because T1 is holding the lock. Possible ways to solve this deadlock are to execute the SQL queries that

access the same set of tables in a fixed order, or add a pessimistic lock (as shown in Listing 4.2). In short, the origins of the SQL queries returned by SLocator may provide developers additional information to investigate the root cause of deadlocks.

> We evaluate SLocator to illustrate its usage in locating the application code that results in generating two cases of problematic SQL queries, i.e., slow SQL queries and SQL queries that cause database deadlocks. We find that SLocator provides developers with additional information to localize the database access issues.

4.5 Threats to Validity

External Validity. We evaluate SLocator exclusively on seven open-source applications implemented using the Hibernate ORM, which may affect the generalizability of our results. To mitigate these threats, we choose the studied applications with various sizes (the LOC ranges from 2.4K to 197K) across different domains such as e-commerce, CMS, and forum to improve the generalizability. Another threat may come from the studied application PetClinic, which only has 2.4K lines of source code. However, we find that the average accuracy does not change much after excluding PetClinic. For example, when using SQL session logs, the average Top@5 for locating the web request changes from 91.7% to 91.2% while the average Top@5 for locating the control flow path changes from 88.8% to 87.7%. Moreover, the approach in SLocator may be applicable to applications using non-Hibernate ORM. Future studies may apply the needed changes to evaluate how SLocator performs on applications implemented using other ORM frameworks.

Construct Validity. One possible threat to construct validity might come from the workload in our experimental setup. We use simulated workload to exercise different workflows in the studied applications. However, this simulated workload may not cover all the workflows and may not be representative of real application workflows. To mitigate these threats, our workload achieves high coverage of the web page and database table, although a high value in different metrics is needed to fully address the construct threat.

A Limitation of static analysis in our approach. One threat is the limitation of static analysis in inferring control flow paths (as discussed in RQ1). For example, due to different used frameworks and the embedded code logic in the user interface (UI), static analysis may not be able to infer the complete code path that generates a SQL query. Future studies need to consider the peculiarity of various web frameworks of applications when analyzing the source code statically. Another limitation may exist in inferring the database access that occurs outside of the web requests. We choose the web request handling methods as the entry points to the studied applications because most functionalities

in a web-based application are accessible through web requests. To verify our design decision, we conduct a backward control flow analysis on the entry points of the database access calls in the studied applications. We found that among all the database accesses, only one database access call in JeeWeb is triggered by a timer in the application instead of accessible through web requests. The limitation of static analysis may also exist in inferring database accesses. Our approach translates the basic and commonly used CRUD operations of the JPA API calls to infer database accesses (templated SQL queries), and extracts the native SQL query and JPQL query as inferred database accesses (inferred queries) (as discussed in Section 4.3.1.2). We did not include criteria APIs due to their dynamic nature. However, we carefully checked the source code of the seven studied applications and found that criteria queries are less used compared to the basic CRUD operations of the JPA EntityManager (there are only 11 usages of Criteria among all the 595 usages of JPA APIs in the studied applications). Future studies should consider examining the usage of various JPA APIs and may expand SLocator's translation layer to cover APIs such as Criteria.

Populated database. We use the synthesized database content to populate the main database tables (as discussed in Section 4.4.1). However, the applications running on the synthesized data may behave differently from the actual deployments, which may affect the execution of the studied applications. To mitigate these threats, we try to populate realistic values into the database. For example, we populate unique email addresses and realistic addresses into customer and address tables in BroadleafCommerce. Besides, all of the synthesized database data still follows the association relationships and database constraints in the database. Hence, the applications should execute well on the synthesized database data. The synthetic database data and data-populating scripts (written in procedures in MySQL) are publicly available [11].

4.6 Conclusion

Object-relational mapping (ORM) frameworks are widely used to abstract database access in database-backed web applications. However, when using ORM, developers do not have full control of how a SQL query is generated. Therefore, given a problematic SQL query, developers may encounter challenges to know how and locate where the SQL query is generated. In this chapter, we propose SLocator, an automated approach to locate the control flow path (i.e., the origin) that generates a given SQL query. SLocator combines static analysis and information retrieval (IR) techniques for locating the origin. We evaluate SLocator on seven open source applications by using two types of DBMS logs: SQL session log and individual query log. SLocator achieves good localization accuracy and has a better localization result compared to the baseline. We also conduct a study to demonstrate how SLocator may be used to locate the database access code that generates

problematic SQL queries (i.e., slow SQL queries and database deadlocks). Our findings show the potential of using IR techniques to help locate database-related issues.

Part III

Conclusion and Future Work

Chapter 5

Thesis Contributions and Future Work

In this chapter, we summarize the research and contribution of this thesis. We also discuss potential future work related to database access in database-backed applications.

5.1 Summary

Despite the widespread use of database-backed applications, the inherent difference between these applications and the underlying DBMS makes database access challenging. Developers may face various challenges in accessing the database when using different technologies. Additionally, due to the abstraction of ORM frameworks, developers may face challenges when debugging database access problems associated with problematic SQL queries. To address these challenges, this thesis aims to understand database access issues and assist in debugging database access issues by locating the associated problematic SQL queries. By manually examining bug reports and commit histories of large-scale Java applications that use SQL queries or ORM frameworks, we investigate and derive the characteristics of database access issues, including categories, root cause, impact, and occurrence. We believe that our findings can be useful for developers to help them avoid pitfalls and serve as a checklist to help testers improve test scenarios that address specific database access bugs. Our results also provide motivations and guidelines for future research to help avoid, detect, and test database access bugs in database-backed applications. Furthermore, we propose an approach for locating the origin (i.e., the control flow path containing a sequence of method calls) that generates a given SQL query. The approach is effective in locating data access issues that generate problematic SQL queries, such as slow SQL queries and SQL queries that cause database deadlocks. The results

show the potential of using IR techniques to help locate database-related issues.

5.2 Thesis Contribution

To address the challenges of database access, this thesis aims to understand the characteristics of database access issues and proposes an approach to locate database issues related to SQL queries. The contributions of this thesis are summarized as follows:

- We empirically study the characteristics of database access bugs in Java applications. We derive five categories (SQL queries, Schema, API, Configuration, SQL query result) of the root causes of database access bugs, containing 25 unique root causes, which can be useful to help developers avoid pitfalls and serve as a checklist to help testers improve test scenarios that address specific database access bugs (Chapter 3).

- To the best of our knowledge, we conduct the first empirical study of database access bugs across JDBC and Hibernate. We find that SQL queries, Schema, and API bugs cover 84.2% of database access bugs across all studied applications. In particular, SQL queries bug (54%) and API bug (38.7%) are the most frequent issues when using JDBC and Hibernate, respectively. The distribution of database access bugs between using JDBC and ORM frameworks provides complementary to developers in selecting database access technologies, which often require trade-offs (Chapter 3).

- We propose SLocator, which leverages both static analysis and information retrieval (IR) techniques to locate where a SQL query is generated in the application code. Our approach outperforms the baseline approach and achieves good localization results for different levels of granularity in SQL queries. For SQL queries in sessions, it achieves a Top@5 accuracy ranging from 78.3% to 95.5% , marking a 225% improvement over the baseline. For individual query logs, the Top@5 accuracy ranges from 59.1% to 100%, marking a 333% improvement compared to the baseline (Chapter 4).

5.3 Future work

This thesis aims to understand the database access bugs during the development of database-backed applications and provides support on locating the SQL queries in the application source code that may cause database access issues. There are still many open challenges and research opportunities that may complement this thesis to improve the quality of database access code or

provide support for developers to test and debug the database access. We highlight some directions for future work.

Supporting for the development of database access code in database-backed applications. In this thesis, we find that database access bugs related to SQL queries or the database schema (e.g., syntax error or inconsistency with the database schema) are the most frequent category of database access bugs when using JDBC. For example, developers may make a typo in the SQL query or fail to construct the search criteria in the SQL query as expected, resulting in problematic SQL queries that are not checked at compile time. We also find that these types of bugs still exist when using Hibernate because developers may need to use HQL/JPQL queries for more complex database access. Therefore, to assist developers improve the quality of database-backed applications, there is a need for better tooling support to verify the SQL queries and database schema. Specifically, future research may work on tools that statically extract the SQL queries (including HQL/JPQL queries) from the application code and verify their syntax. The tools can also verify the consistency between the database schema and the SQL queries extracted from the the application code. In this way, these tools statically detect errors in SQL queries and will alleviate more SQL query bugs during system development.

Supporting for the maintenance of database access code in database-backed applications. Future research may develop new approaches to automatically maintain database access code. During the development of database-backed applications, both the database schema and the ORM-related code evolves frequently [125, 42]. However, this evolution of database-backed applications and the underlying database can lead to inconsistencies between the SQL query and the database schema. For example, in this thesis we find that the table/column specified in the SQL query may be deleted, renamed, or not yet created in the database, leading to database access bugs. Therefore, developers may benefit from approaches that automatically suggest updates to the database schema or database access code when developers modify the code. For example, when a developer modifies the database schema, the approach can automatically identify all the database access code (e.g., either SQL queries or ORM database access APIs) that is affected by the change and needs to be updated.

Examining the Adequacy of Database Access Testing in Large-Scale Database-Backed Applications. In this thesis, we investigate the root cases of database access bugs and find that some database access bugs occur when the database access code is not fully covered by the test cases. However, there is a lack of study on the adequacy of database access testing in large-scale database-backed applications. In the future, we can further investigate the open source database-backed applications by examining what percentage of database access methods are covered by the tests and what types of database access methods are prone to be missed by the tests. For those

database access methods associated with tests, we can further examine their line coverage and branch coverage. If the lines of the database access methods are not fully covered, we can examine what specific database access source code (e.g., building query criteria) is missed. For the uncovered specific types of database access methods and lines of the database access source code, developers can focus on them when writing database access tests, and researchers can work on automatic tools to generate adequate tests for them.

Examining the Effectiveness of Automatic Test Generation on Database Access Code. Unit tests play a key role in ensuring program correctness. However, writing unit tests manually is a time-consuming and laborious task. Automated test generation techniques, such as the state-of-the-art search-based software testing tool EvoSuite [58], have shown their effectiveness. However, these techniques may be hampered when generating tests for database access code. Database access code may not be fully covered because it depends on the state of the database. For example, the database access code may have different paths to handle the query results. If the database has no corresponding data row, it will not return a query result, and the corresponding path may not be covered. In future studies, we may seek to investigate the effectiveness of search-based test generation, machine-based test generation, and large-language model-based test generation on the database access code. Specifically, we will focus on the correctness, coverage, and readability of the automatically generated tests.

Bibliography

[1] Django. URL https://www.djangoproject.com/.

[2] Pep 249 – python database api specification v2.0. URL https://peps.python.org/pep-0249/.

[3] A modern, simple and very fast mysql library for ruby, . URL https://github.com/brianmario/mysql2.

[4] Ruby on rails, . URL https://rubyonrails.org/.

[5] Sqllint - detecting semantic errors in sql queries. URL https://dbs.informatik.uni-halle.de/sqllint/.

[6] channable/dbcritic. https://github.com/channable/dbcritic. (Accessed on 03/25/2023).

[7] Db optimization service - holistic.dev. https://holistic.dev/. (Accessed on 03/25/2023).

[8] What java orm do you prefer, and why?, 2009. URL https://stackoverflow.com/questions/452385/what-java-orm-do-you-prefer-and-why.

[9] Jpa or jdbc, how are they different?, 2012. URL https://stackoverflow.com/questions/11881548/jpa-or-jdbc-how-are-they-different/.

[10] Replication package, 2021. URL https://github.com/SPEAR-SE/empirical-db-issue-data.

[11] Online appendix/replication package for "SLocator: Localizing the origin of sql queries in database-backed web applications", 2021. URL https://github.com/liuwei-tianshu/SLocator.

[12] Eager fetching is a code smell when using jpa and hibernate. https://vladmihalcea.com/eager-fetching-is-a-code-smell/, 2021. Last accessed Aug. 2021.

[13] Spring framework. https://spring.io/, 2021. Last accessed Aug. 2021.

[14] Usage statistics of site elements for websites, 2022. URL `https://w3techs.com/technologies/overview/site_element`.

[15] Pypl popularity of programming language index. `https://pypl.github.io/PYPL.html`, 2022. (Accessed on 12/21/2022).

[16] ADempiere. Adempiere business suite, 2021. URL `https://github.com/adempiere/adempiere`.

[17] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. Students' semantic mistakes in writing seven different types of sql queries. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 272–277, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342315.

[18] Bader Alshemaimri, Ramez Elmasri, Tariq Alsahfi, and Mousa Almotairi. A survey of problematic database code fragments in software systems. *Engineering Reports*, 3(10):e12441, 2021.

[19] David Anderson. Modeling and analysis of sql queries in php systems. Master's thesis, East Carolina University, April 2018.

[20] Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, and Varmo Vene. An interactive tool for analyzing embedded sql queries. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 131–138, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17164-2.

[21] Andrea Arcuri. RESTful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), jan 2019. ISSN 1049-331X.

[22] Andrea Arcuri and Juan P. Galeotti. Handling sql databases in automated system test generation. *ACM Trans. Softw. Eng. Methodol.*, 29(4), jul 2020. ISSN 1049-331X.

[23] N. Arzamasova, K. Böhm, B. Goldman, C. Saaler, and M. Schäler. On the usefulness of sql-query-similarity measures to find user interests. *IEEE Transactions on Knowledge and Data Engineering*, 32(10):1982–1999, Oct 2020. ISSN 1558–2191. doi: 10.1109/TKDE.2019.2913381.

[24] Natalia Arzamasova, Martin Schäler, and Klemens Böhm. Cleaning antipatterns in an sql query log. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1751–1752, 2018.

[25] bbs, 2022. URL `https://github.com/diyhi/bbs`.

[26] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939951. doi: 10.1145/1453101.1453146.

[27] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 142–153, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338970.

[28] S. Brass and C. Goldberg. Semantic errors in sql queries: a quite complete list. In *Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings.*, pages 250–257, 2004.

[29] BroadleafCommerce. Broadleafcommerce - enterprise ecommerce framework based on spring, 2021. URL `https://github.com/BroadleafCommerce/BroadleafCommerce`.

[30] JRebel by Perforce. Java tools and technologies landscape 2014, 2014. URL `https://www.jrebel.com/resources/java-tools-and-technologies-landscape-2014`.

[31] Vasco Ferreira C. Hibernate debugging - finding the origin of a query, 2022. URL `https://dzone.com/articles/hibernate-debugging-where-does`.

[32] George G. Cabral, Leandro L. Minku, Emad Shihab, and Suhaib Mujahid. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 666–676. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00076.

[33] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based test data generation for sql queries. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1220–1230, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381.

[34] W.K. Chan, S.C. Cheung, and T.H. Tse. Fault-based testing of database application programs with conceptual data model. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 187–196, 2005.

[35] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi: 10.1109/TSE.2021.3071473.

[36] Boyuan Chen, Zhen Ming (Jack) Jiang, Paul Matos, and Michael Lacaria. 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, page 653–664. IEEE Press, 2020. ISBN 9781728125084. doi: 10.1109/ASE.2019.00066.

[37] Tse-Hsun Chen. *Improving the performance of database-centric applications through program analysis*. PhD thesis, Queen's University (Canada), 2016.

[38] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1001–1012, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565.

[39] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 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, pages 666–677, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186.

[40] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting problems in the database access code of large scale systems - an industrial experience report. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 71–80, 2016.

[41] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 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):1148–1161, 2016.

[42] Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 165–176, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868.

[43] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

[44] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. *SIGPLAN Not.*, 48(6):3–14, jun 2013. ISSN 0362-1340.

[45] CloudStore, 2022. URL `https://github.com/CloudScale-Project/CloudStore`.

[46] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[47] C. Coronel and S. Morris. *Database Systems: Design, Implementation, & Management*, chapter 15. Cengage Learning, 13 edition, 2018. ISBN 9781337627900.

[48] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Empirical comparison of black-box test case generation tools for restful apis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 226–236, 2021. doi: 10.1109/SCAM52516.2021.00035.

[49] Carlo Curino, Hyun J. Moon, Letizia Tanca, and Carlo Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In José Cordeiro and Joaquim Filipe, editors, *ICEIS (1)*, pages 3231–332, 2008. ISBN 978-989-8111-36-4.

[50] Christine Dancey. *Statistics without maths for psychology.* Pearson/Prentice Hall, Harlow, England New York, 2007. ISBN 978-0-13-205160-6.

[51] DBeaver. Free universal database tool and sql client, 2021. URL `https://github.com/dbeaver/dbeaver`.

[52] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. Sqlcheck: Automated detection and diagnosis of sql anti-patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2331–2345, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356.

[53] dotCMS. Source code for dotcms hybrid content management system, 2021. URL `https://github.com/dotCMS/core`.

[54] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937346.

[55] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. Combining query reduction and expansion for text-retrieval-based bug localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 166–176, 2021. doi: 10.1109/SANER50967.2021.00024.

[56] Apache Software Foundation. Apache jmeter, 2021. URL `https://jmeter.apache.org/`.

[57] Eclipse Foundation. The aspectj project, 2021. URL `https://www.eclipse.org/aspectj/`.

[58] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179.

[59] GitHub. The top programming languages, 2022. URL `https://octoverse.github.com/2022/top-programming-languages`. Last accessed Nov. 2022.

[60] Francisco Gonçalves de Almeida Filho, Antônio Diogo Forte Martins, Tiago da Silva Vinuto, José Maria Monteiro, Ítalo Pereira de Sousa, Javam de Castro Machado, and Lincoln Souza Rocha. Prevalence of bad smells in pl/sql projects. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 116–121, 2019. doi: 10.1109/ICPC.2019.00025.

[61] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, page 645–654, USA, 2004. IEEE Computer Society. ISBN 0769521630.

[62] Mark Grechanik, B. M. Mainul Hossain, Ugo Buy, and Haisheng Wang. Preventing database deadlocks in applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 356–366, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491412.

[63] Bhanu Pratap Gupta, Devang Vira, and S. Sudarshan. X-data: Generating test data for killing sql mutants. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 876–879, 2010.

[64] Sonia Haiduc. Automatically detecting the quality of the query and its implications in ir-based concept location. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 637–640, 2011. doi: 10.1109/ASE.2011.6100144.

[65] Sonia Haiduc and Andrian Marcus. On the effect of the query in ir-based concept location. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 234–237, 2011. doi: 10.1109/ICPC.2011.48.

[66] William G.J. Halfond and Alessandro Orso. Command-form coverage for testing database applications. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 69–80, 2006.

[67] Hibernate. Built-in constraints, 2021. URL `https://docs.jboss.org/hibernate/validator/6.0/reference/en-US/html_single/#section-builtin-constraints`.

[68] Hibernate. Hql and jpql, 2021. URL `https://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html#hql`.

[69] Hibernate. Logging, 2021. URL `https://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html#best-practices-logging`. Last accessed Jul. 2021.

[70] Hibernate. Logging. `https://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html#best-practices-logging`, 2021. Last accessed Aug. 2021.

[71] Zijie Huang, Zhiqing Shao, Guisheng Fan, Huiqun Yu, Kang Yang, and Ziyi Zhou. Hbsniff: A static analysis tool for java hibernate object-relational mapping code smell detection. *Sci. Comput. Program.*, 217(C), may 2022. ISSN 0167-6423.

[72] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1110–1121, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216.

[73] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Discov. Data*, 2(2), July 2008. ISSN 1556-4681.

[74] JeeWeb, 2022. URL `https://github.com/white-cat/jeeweb`.

[75] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[76] Gregory M. Kapfhammer, Phil McMinn, and Chris J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *2013 IEEE*

Sixth International Conference on Software Testing, Verification and Validation, pages 31–40, 2013.

[77] B. Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 2010. ISBN 9781934356555.

[78] Mike Keith, Merrick Schincariol, and Massimo Nardone. *Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs*, chapter 4, pages 101–155. Apress, Berkeley, CA, 3 edition, 2018. ISBN 978-1-4842-3420-4. doi: 10.1007/978-1-4842-3420-4_4.

[79] G. Kul, D. T. A. Luong, T. Xie, V. Chandola, O. Kennedy, and S. Upadhyaya. Similarity metrics for sql query clustering. *IEEE Transactions on Knowledge and Data Engineering*, 30 (12):2408–2420, Dec 2018. ISSN 1558–2191. doi: 10.1109/TKDE.2018.2831214.

[80] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977. ISSN 0006341X, 15410420.

[81] Tien-Duy B. Le, Ferdian Thung, and David Lo. Predicting effectiveness of ir-based bug localization techniques. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 335–345, 2014. doi: 10.1109/ISSRE.2014.39.

[82] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: Reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 61–72, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213856.

[83] Wei Liu and Tse-Hsun Chen. Slocator: Localizing the origin of sql queries in database-backed web applications. *IEEE Transactions on Software Engineering*, 49(6):3376–3390, 2023. doi: 10.1109/TSE.2023.3253700.

[84] Wei Liu, Shouvick Mondal, and Tse-Hsun (Peter) Chen. An empirical study on the characteristics of database access bugs in java applications. *ACM Trans. Softw. Eng. Methodol.*, 33(7), September 2024. ISSN 1049-331X. doi: 10.1145/3672449.

[85] Yingjun Lyu, Ali Alotaibi, and William G. J. Halfond. Quantifying the performance impact of SQL antipatterns on mobile applications. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 53–64, 2019.

[86] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. Sand: A static analysis approach for detecting sql antipatterns. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 270–282, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384599.

[87] ManageForce. System failure - the cost of database downtime, 2016. URL `http://www.manageforce.com/blog/dba-suffering-from-system-failure-infographic`. Last accessed Nov. 2021.

[88] Petros Manousis, Apostolos Zarras, Panos Vassiliadis, and George Papastefanatos. Extraction of embedded queries via static analysis of host code. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, pages 511–526, Cham, 2017. Springer International Publishing.

[89] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. Impact analysis of database schema changes. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 451–460, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791.

[90] Phil Mcminn, Chris J. Wright, and Gregory M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Trans. Softw. Eng. Methodol.*, 25(1), dec 2015. ISSN 1049-331X.

[91] Phil McMinn, Chris J. Wright, Colton J. McCurdy, and Gregory M. Kapfhammer. Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas. *IEEE Transactions on Software Engineering*, 45(5):427–463, 2019.

[92] metasfresh. We do open source erp - fast, flexible & free software to scale your business., 2021. URL `https://github.com/metasfresh/metasfresh`.

[93] Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of dynamic database usage in java systems. In Selmin Nurcan, Pnina Soffer, Marko Bajec, and Johann Eder, editors, *Advanced Information Systems Engineering*, pages 491–506, Cham, 2016. Springer International Publishing.

[94] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 262–273, 2016.

[95] Microsoft. getmoreresults skips resultsets[bug] #969, 2019. URL `https://github.com/microsoft/mssql-jdbc/issues/969`.

[96] Vlad Mihalcea. hypersistence-utils. URL `https://github.com/vladmihalcea/hypersistence-utils`. Last accessed Nov. 2022.

[97] Vlad Mihalcea. How do lockmodetype.pessimistic_read and lockmodetype.pessimistic_write work in jpa and hibernate, 2019. URL `https://vladmihalcea.com/hibernate-locking-patterns-how-do-pessimistic_read-and-pessimistic_write-work/`.

[98] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering*, 25, 09 2020. doi: 10.1007/s10664-020-09823-w.

[99] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 188–197, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00028.

[100] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the prevalence, impact, and evolution of sql code smells in data-intensive systems. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 327–338, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177.

[101] MySQL. Mysql connectors, 2022. URL `https://www.mysql.com/products/connector/`.

[102] Csaba Nagy and Anthony Cleve. Mining stack overflow for discovering error patterns in sql queries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 516–520, 2015. doi: 10.1109/ICSM.2015.7332505.

[103] Csaba Nagy and Anthony Cleve. 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)*, pages 147–152, 2017.

[104] Csaba Nagy and Anthony Cleve. Sqlinspect: A static analyzer to inspect database usage in java applications. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 93–96, 2018.

[105] Csaba Nagy and Anthony Cleve. Sqlinspect: A static analyzer to inspect database usage in java applications. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 93–96, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356633.

[106] Csaba Nagy, Loup Meurice, and Anthony Cleve. Where was this sql query executed? a static concept location approach. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 580–584, 2015. doi: 10.1109/SANER.2015.7081881.

[107] Openfire. An xmpp server licensed under the open source apache license., 2021. URL `https://github.com/igniterealtime/Openfire`.

[108] OpenMRS. Openmrs api and web application code, 2021. URL `https://github.com/openmrs/openmrs-core`.

[109] Oracle. Jax-rs: Java api for restful web services, 2013. URL `https://download.oracle.com/otn-pub/jcp/jaxrs-2_0_rev_A-mrel-eval-spec/jsr339-jaxrs-2.0-final-spec.pdf`. Last accessed Jul. 2021.

[110] Oracle. Lockmodetype, 2015. URL `https://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html`.

[111] Oracle. Jsr 338: Java persistence api, version 2.2, 2017. URL `https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-eval-spec/JavaPersistence.pdf`. Last accessed Jul. 2021.

[112] Oracle. An innodb deadlock example, 2020. URL `https://dev.mysql.com/doc/refman/5.6/en/innodb-deadlock-example.html`.

[113] Oracle. Deadlocks in innodb, 2020. URL `https://dev.mysql.com/doc/refman/5.6/en/innodb-deadlocks.html`.

[114] Oracle. Mysql server logs, 2020. URL `https://dev.mysql.com/doc/refman/5.6/en/server-logs.html`.

[115] Oracle. The general query log, 2020. URL `https://dev.mysql.com/doc/refman/5.6/en/query-log.html`.

[116] Oracle. The slow query log, 2020. URL `https://dev.mysql.com/doc/refman/5.6/en/slow-query-log.html`.

[117] Oracle. Mysql, 2021. URL `https://www.mysql.com/`.

[118] Oracle. Java software, 2021. URL `https://www.oracle.com/java/`. Last accessed Nov. 2021.

[119] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Trans. Softw. Eng. Methodol.*, 23(2), apr 2014. ISSN 1049-331X.

[120] PetClinic. A sample spring-based application, 2022. URL https://github.com/spring-projects/spring-petclinic.

[121] PostgreSQL. Error reporting and logging, 2020. URL https://www.postgresql.org/docs/13/runtime-config-logging.html.

[122] PostgreSQL. Software catalogue - drivers and interfaces, 2022. URL https://www.postgresql.org/download/products/2-drivers-and-interfaces/.

[123] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. Scaffle: Bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 225–236, 2020.

[124] PublicCMS, 2022. URL https://github.com/sanluan/PublicCMS.

[125] Dong Qiu, Bixin Li, and Zhendong Su. 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 2013, pages 125–135, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379.

[126] Redgate. 119 sql code smells. URL https://www.red-gate.com/library/119-sql-code-smells.

[127] Tanmoy Sarkar. *Testing database applications using coverage analysis and mutation analysis.* PhD thesis, Iowa State University, 2013.

[128] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. Database-access performance antipatterns in database-backed web applications. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 58–69, 2020.

[129] Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. Smelly relations: Measuring and understanding database schema quality. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 55–64, 2018.

[130] Ravjot Singh, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 309–320, 2016.

[131] S. Sinha and M.J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000. doi: 10.1109/32.877846.

[132] SlashData. State of the developer nation, 2022. URL `https://slashdata-website-cms.s3.amazonaws.com/sample_reports/VZtJWxZw5Q9NDSAQ.pdf`.

[133] J. Sohn, Y. Kamei, S. McIntosh, and S. Yoo. Leveraging fault localisation to enhance defect prediction. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–294, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society. doi: 10.1109/SANER50967.2021.00034.

[134] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. The significance of bug report elements. *Empirical Software Engineering*, 25:1–40, 11 2020.

[135] María José Suárez-Cabal and Javier Tuya. Using an sql coverage measurement for testing database applications. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138555.

[136] Ferdian Thung, Tien-Duy B. Le, Pavneet Singh Kochhar, and David Lo. Buglocalizer: Integrated tool support for bug localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 767–770, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2661678.

[137] TPC-W, 2022. URL `https://www.tpc.org/tpcw/`.

[138] Javier Tuya, Ma Jose Suarez-Cabal, and Claudio de la Riva. SQLMutation: A tool to generate mutants of sql database queries. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pages 1–1, 2006.

[139] Javier Tuya, M José Suárez-Cabal, and Claudio de la Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007. ISSN 0950-5849.

[140] Javier Tuya, María José Suárez-Cabal, and Claudio de la Riva. Full predicate coverage for testing sql database queries. *Softw. Test. Verif. Reliab.*, 20(3):237–288, sep 2010. ISSN 0960-0833.

[141] WallRide. Multilingual easy-to-customize open source cms made by java, 2022. URL `https://github.com/tagbangers/wallride`.

[142] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *J. Softw. Evol. Process*, 28(10):921–942, October 2016. ISSN 2047–7473. doi: 10.1002/smr.1801.

[143] Shaowei Wang, David Lo, and Julia Lawall. Compositional vector space models for improved bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 171–180, 2014. doi: 10.1109/ICSME.2014.39.

[144] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 286–300, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127.

[145] Todd Warszawski and Peter Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 5–20, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064037.

[146] M. Wen, R. Wu, and S. Cheung. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–273, 2016.

[147] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190, 2014. doi: 10.1109/ICSME.2014.40.

[148] Chris J. Wright, Gregory M. Kapfhammer, and Phil McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *2014 14th International Conference on Quality Software*, pages 57–66, 2014.

[149] Xin Xia, David Lo, Xingen Wang, Chenyi Zhang, and Xinyu Wang. Cross-language bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 275–278, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597788.

[150] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 1299–1308, 2017. ISBN 9781450349185.

[151] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 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, pages 800–810, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381.

[152] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 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, page 884–887, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3264589. URL `https://doi.org/10.1145/3236024.3264589`.

[153] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. View-centric performance optimization for database-backed web applications. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 994–1004, 2019.

[154] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. *Managing Data Constraints in Database-Backed Web Applications*, pages 302–303. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371223.

[155] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGPLAN Not.*, 45(3):143–154, March 2010. ISSN 0362–1340. doi: 10.1145/1735971.1736038.

[156] Lei Zeng, Yang Xiao, and Hui Chen. Linux auditing: Overhead and adaptation. In *2015 IEEE International Conference on Communications (ICC)*, pages 7168–7173, 2015. doi: 10.1109/ICC.2015.7249470.

[157] Chixiang Zhou and Phyllis Frankl. Jdama: Java database application mutation analyser. *Softw. Test. Verif. Reliab.*, 21(3):241–263, sep 2011. ISSN 0960-0833.

[158] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012. doi: 10.1109/ICSE.2012.6227210.

[159] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the*

37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 415–425. IEEE Press, 2015. ISBN 9781479919345.

[160] Daniel Zwillinger and Stephen Kokoska. *CRC standard probability and statistics tables and formulae*. Chapman & Hall/CRC, Boca Raton, 2000. ISBN 9781584880592.