Improving the Performance of Database-Centric Applications Through Program Analysis

by

Tse-Hsun Chen

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

September 2016

Abstract

MODERN software applications are becoming more dependent on database management systems (DBMSs). DBMSs are usually used as black boxes by software developers. For example, Object-Relational Mapping (ORM) is one of the most popular database abstraction approaches that developers use nowadays. Using ORM, objects in Object-Oriented languages are mapped to records in the database, and object manipulations are automatically translated to SQL queries. As a result of such conceptual abstraction, developers do not need deep knowledge of databases; however, all too often this abstraction leads to inefficient and incorrect database access code. Thus, this thesis proposes a series of approaches to improve the performance of database-centric software applications that are implemented using ORM. Our approaches focus on troubleshooting and detecting inefficient (i.e., performance problems) database accesses in the source code, and we rank the detected problems based on their severity. We first conduct an empirical study on the maintenance of ORM code in both open source and industrial applications. We find that ORM performance-related configurations are

rarely tuned in practice, and there is a need for tools that can help improve/tune the performance of ORM-based applications. Thus, we propose approaches along two dimensions to help developers improve the performance of ORM-based applications: 1) helping developers write more performant ORM code; and 2) helping developers configure ORM configurations.

To provide tooling support to developers, we first propose static analysis approaches to detect performance anti-patterns in the source code. We automatically rank the detected anti-pattern instances according to their performance impacts. Our study finds that by resolving the detected anti-patterns, the application performance can be improved by 34% on average. We then discuss our experience and lessons learned when integrating our anti-pattern detection tool into industrial practice. We hope our experience can help improve the industrial adoption of future research tools. However, as static analysis approaches are prone to false positives and lack runtime information, we also propose dynamic analysis approaches to further help developers improve the performance of their database access code. We propose automated approaches to detect redundant data access anti-patterns in the database access code, and our study finds that resolving such redundant data access anti-patterns can improve application performance by an average of 17%. Finally, we propose an automated approach to tune performance-related ORM configurations using both static and dynamic analysis. Our study shows that our approach can help improve application throughput by 27–138%.

Through our case studies on real-world applications, we show that all of our proposed approaches can provide valuable support to developers and help improve application performance significantly.

Acknowledgments

First of all, I would like to thank my supervisor Dr. Ahmed E. Hassan for his continuous guidance and support throughout my graduate study. You are the best mentor and supervisor that one can ask for. I am extremely lucky that I have the chance to work under your supervision. Thank you very much for encouraging me to pursue the research topics that I love. Your great guidance has helped me grow not only as a researcher, but also a thinker. Your advice has become my mottos and is priceless to my future career.

A sincere appreciation to my supervisory and examination committee members, Dr. Patrick Martin, Dr. Juergen Dingel, Dr. Hossam Hassanein, and Dr. Mark Grechanik for their continued critique and guidance. Many thanks to my examiners for their valuable and insightful feedback on my work.

I am very honored to have the chance to work and collaborate with the brightest researchers during my Ph.D. career. I would like to thank all of my labmates and collaborators, Dr. Weiyi Shang, Dr. Zhen Mining Jiang, Dr. Meiyappan Nagappan, Heng Li, Dr. Emad Shihab, Dr. Stephen Thomas, Dr. Hadi Hemmati, and Dr.

Michael Godfrey for all the fruitful discussions and collaborations.

I would like to thank BlackBerry and especially the members of the Performance Engineering team. I cloud not find my thesis topic without the industrial environment and thoughtful feedback provided by the BlackBerry team. I would like to thank Mohamed Nasser and Parminder Flora for their continuous support throughout my Ph.D. career.

A special thanks to my family, especially my father, mother, and brother. Thank you for all the sacrifices that you have made for me during my study. Your precious and invaluable love have been my greatest support in my life. At the end, I would like to express my greatest appreciation to my beloved wife Jinqiu Yang. Jinqiu, thank you very much for your understanding and continuous support. I dedicate this thesis to you.

Dedication

To my parents, my family, and my special lady, Jinqiu Yang.

Related Publications

In all chapters and related publications of the thesis, my contributions are: drafting the initial research idea; researching background knowledge and related work; implementing the tools; conducting experiments; and writing and polishing the writing. My co-authors supported me in refining the initial ideas, pointing me to missing related work, providing feedback on earlier drafts, and polishing the writing.

Earlier versions of the work in the thesis were published as listed below:

1. *Improving the Quality of Large-Scale Database-Centric Software Systems by Analyzing Database Access Code (Chapter 1)*
   Tse-Hsun Chen, 31st International Conference on Data Engineering (ICDE), PhD Symposium, 2015. Seoul, Korea. Pages 245–249.

2. *An Empirical Study on the Practice of Maintaining Object-Relational Mapping Code in Java Systems (Chapter 4)*
   Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora, 13th International Conference on

Mining Software Repositories (MSR), 2016. Austin, Texas. Pages 165–176.

3. *Detecting Performance Anti-patterns for Applications Developed using Object-relational Mapping (Chapter 5)*
   Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora, 36th International Conference on Software Engineering (ICSE), 2014. Hyderabad, India. Pages 1001–1012.

4. *Detecting Problems in Database Access Code of Large Scale Systems - An Industrial Experience Report (Chapter 6)*
   Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora, 38th International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP), 2016. Austin, Texas. Pages 71–80.

5. *Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks (Chapter 7)*
   Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora, IEEE Transactions on Software Engineering (TSE), 2016. In Press.

6. *CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-based Database-centric Web Applications (Chapter 8)*
   Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora, 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2016. Seattle, WA. Accepted.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

An earlier version of this chapter is published at the 31st International Conference on Data Engineering (ICDE), PhD Symposium, 2015. Seoul, Korea. Pages 245–249. (Chen, 2015b)

MODERN software applications are nowadays more dependent on the underlying database management system (DBMS) for data integrity and management. Developers often store all user data in DBMSs to provide better scalability and maintainability. Thus, DBMSs are one of the core components in these database-centric applications. Although DBMSs are usually fairly optimized in terms of performance and data management, how developers control and communicate with the DBMS has a significant impact on the performance

of database-centric software applications. Since managing the data consistency between the source code and the DBMS is a difficult task, especially for complex large-scale applications, several frameworks have been proposed to ease and abstract the complexity of data access. For example, developers often employ Object-Relational Mapping (ORM) frameworks to provide a conceptual abstraction between objects in Object-Oriented Programming Languages and records in the underlying DBMS. Using ORM frameworks, changes in the object states are propagated automatically to the corresponding records in the DBMS. These abstraction frameworks significantly reduce the amount of code that developers need to write (Barry and Stanienda, 1998; Leavitt, 2000); however, due to the black box nature of such abstraction layers, developers may not fully understand the behaviour of the framework-generated SQL queries, which may result in performance problems. In short, despite the promoted benefits of these frameworks, using them incorrectly can lead to poor application quality.

1.1 Thesis Statement

Existing approaches primarily improve application performance from a database perspective, e.g., SQL query optimization. However, all too often, the root cause of such problematic queries is having inefficient database access code. Recent studies (e.g., Chavan *et al.* (2011b); Cheung *et al.* (2013a, 2014)) propose various frameworks on top of database access abstraction layers (e.g., ORM) to automatically transform and optimize the framework-generated SQL queries. However, a potential problem with these approaches is that the database-centric application may experience higher overheads and become harder to debug due to the extra layer

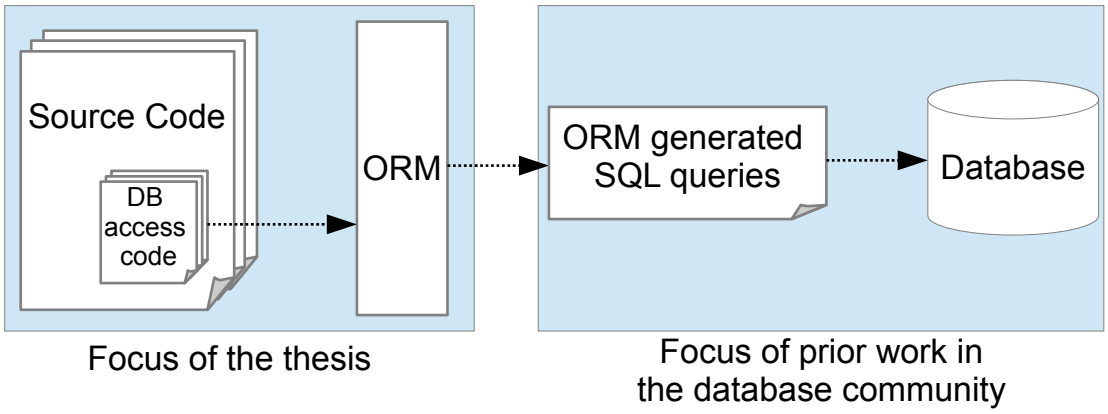


Figure 1.1: Focus of the thesis.

of complexity. We, on the other hand, believe that by identifying problems or possible places for improvement in the database access code, developers can allocate performance improvement resources more effectively and directly. Therefore, we propose:

> *The key to improving the performance of database-centric applications is not only by improving the backend database management system, but also by improving the database access code, which is rarely considered in prior studies.*

In this thesis, we consider performance problems as potential places in the code that need performance optimization (e.g., increase throughput or decrease response time), since not all performance problems have a non-trivial impact in practice.

1.2 Thesis Overview

Figure 1.1 gives an overview of the focus of this thesis, and how it differs from prior studies. We focus on ORM-based database access code, since ORM frameworks are

widely used in practice (e.g., more than 67% of Java developers use ORM (ZeroturnAround, 2014)). We now give an overview of the work that is presented in this thesis.

1.2.1 Chapter 2: Literature Review of Applying Program Analysis to Improve Database Access Code

For our literature review, we focus on prior studies that attempt to improve or detect problems in database access code. We characterize and compare the surveyed literature along the following dimensions:

1. Program analysis: We report on prior studies that use static or dynamic analysis to detect problems in database access code;
2. Code transformation: We report on prior studies that improve database access code by statically or dynamically transforming database access code;
3. Domain specific languages and APIs: We report on prior studies that propose domain specific languages and APIs for improving database access code.

From our literature review, we find that there is lack of tooling support for detecting performance problems in the database-access code. We also observe that most prior studies do not provide an automated evaluation or prioritization of the detected problems in the database access code, and most prior studies do not consider how the queried data would be used in the application.

1.2.2 Chapter 3: Background

We provide a brief background of how developers use ORM frameworks to access the DBMS. ORM frameworks abstract SQL queries as a set of API calls along with associated configurations. We first provide a brief overview of different ORM frameworks, then we discuss how ORM frameworks ease access of the DBMS.

1.2.3 Chapter 4: Maintenance Activities of ORM Code

Despite the advantages of using ORM frameworks, we observe several difficulties in maintaining ORM code when cooperating with our industrial partner. After conducting studies on other open source applications, we find that such difficulties are common in other Java applications. In particular, we find that i) ORM cannot completely encapsulate database accesses in objects or abstract the underlying database technology, thus leading ORM code changes to be more scattered; ii) ORM code changes are more frequent than regular code, but there is a lack of tools that help developers verify the correctness of ORM code at compilation time; iii) we find that changes to ORM code are more commonly due to performance or security reasons; however, traditional static code analyzers need to be extended to capture the peculiarities of ORM code in order to detect such problems. In short, we highlight the hidden maintenance costs of using ORM frameworks, and provide some initial insights about potential approaches to help maintain ORM code.

1.2.4 Chapter 5: Statically Detecting ORM Performance Anti-patterns

We propose an automated framework to detect ORM performance anti-patterns in the source code. Furthermore, as there could be hundreds or even thousands of instances of anti-patterns, our framework is able to prioritize the detected anti-patterns based on a statistically rigorous performance assessment of their impact. We have successfully evaluated our framework on one open source application and another large-scale industrial applications. Our evaluation demonstrates that our framework can detect new and known real-world performance bugs and that fixing the detected instances of performance anti-patterns can improve the application response time by up to 69%.

1.2.5 Chapter 6: Adopting Anti-pattern Detection Framework in Practice

We document our industrial experience over the past few years on detecting anti-patterns in database access code, implementing an anti-pattern detection tool, and integrating the tool into daily practice. We discuss the challenges that we encountered and the lessons that we learned during integrating our tool into practice. Since most applications nowadays are leveraging frameworks, we also provide a detailed discussion of five framework-specific database access anti-patterns that we found. We hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language anti-patterns and associated detectors.

1.2.6 Chapter 7: Dynamically Detecting Redundant Data Anti-patterns

We propose an automated approach, which we implement as a Java framework, to dynamically detect redundant data anti-patterns. We apply our framework on one enterprise application and two open source applications. Our analysis finds that redundant data anti-patterns exist in 87% of the exercised transactions. Due to the large number of detected redundant data anti-patterns, we propose an automated approach to assess the impact and prioritize the resolution efforts. Our performance assessment result shows that by resolving the redundant data anti-patterns, the application response time for the studied applications can be improved by an average of 17%.

1.2.7 Chapter 8: Automated ORM Cache Configuration Tuning

We propose CacheOptimizer, a lightweight approach that helps developers optimize the configuration of caching frameworks for web applications that are implemented using Hibernate (one of the most popular ORM frameworks). CacheOptimizer leverages readily-available web logs to create mappings between a workload and database access requests. Given the mappings, CacheOptimizer discovers the optimal cache configuration using coloured Petri nets, and automatically adds the appropriate cache configurations to the application. We evaluate CacheOptimizer on three open-source web applications. We find that i) CacheOptimizer improves the throughput by 27–138%; and ii) after considering both the memory cost and throughput improvement, CacheOptimizer still brings statistically significant gains

(with mostly large effect sizes) in comparison to the developers' default cache configuration and blindly enabling all possible caches.

1.3 Thesis Contributions

In this thesis, we demonstrate that, in order to improve the performance of database-centric applications, it is important to help developers write better database access code. In addition, we show that it is valuable to focus not only on the traditional SQL queries, since developers nowadays usually leverage different frameworks to access the database. In particular, our contributions are as follows:

1. We show that performance problems are more likely to occur in database access code, but developers rarely tune performance-related configurations for database accesses.

2. We propose an automated approach that finds the optimal performance configuration for database-centric applications.

3. We demonstrate that many performance-related problems in the database access code can be found prior to release using automated anti-pattern detectors.

4. We provide an experience report on adopting our static analysis framework into practice, and we hope our experience can assist others who wish to deploy their research in practice.

CHAPTER 2

Literature Review of Applying Program Analysis to Improve Database Access Code

In this thesis, we propose approaches to help developers improve database access code, with a focus on ORM code. Most prior studies in the database community usually focus on implementing frameworks that can either statically or dynamically transform the database access code into a more optimized form (i.e., more efficient SQL queries). However, such approaches are difficult to debug due to the added complexity. In addition, these frameworks may cause extra overhead to the application, and the framework implementations are usually not available to developers.

This thesis aims to improve the database access code from the perspective of Software Engineering research, where we try to identify and rank problems (i.e., anti-patterns) in the code, and assist developers in fixing the problems. Thus, the applications can be easier to maintain, and developers have full control of how they want to resolve a problem.

For our literature review, we focus on prior studies that attempt to improve or detect problems in database access code. Note that we exclude security-related studies in our review, since this thesis mostly focuses on performance-related problems in database access code, and security problems are a different area of study. We characterize and compare the surveyed literature along the following dimensions:

- **Program analysis**: We report on studies that use static or dynamic analysis to detect problems in database access code.
- **Code transformation**: We report on studies that improve database access code by statically or dynamically transforming database access code.

- **Domain specific languages and APIs**: We report on studies that propose domain specific languages and APIs for improving database access code.

2.1 Paper Selection Process

There exists a large amount of prior work that focuses on improving the quality of DBMS, but not as much work that attempts to improve the quality of database access code. Since improving application qualities by analyzing database access code is an interdisciplinary area of study that involves database and software engineering research, we survey the papers in these two areas of computing. Below, we briefly explain the paper selection process for our literature review.

Table 2.1 lists the venues that we surveyed. We consider the papers that are published in the past 10 years (2005 – 2015). To improve the coverage of our literature review, we also check the citations of each relevant paper. In total, we surveyed 40 papers in the areas of database and software engineering.

2.2 Program Analysis

Table 2.2 shows a list of popular static anti-pattern detection tools. FindBugs (Hovemeyer and Pugh, 2004) is a widely-used open source static Java anti-pattern detection tool. FindBugs scans Java binaries to detect general bugs, bad coding styles, and some security problems. PMD (PMD, 2016) is a static source code analyzer that detects potential problems in the code using pre-defined rules that are related to general coding problems (e.g., empty code blocks and bad code design). Error Prone (Google, 2016) is a code analysis tool that is developed by Google. The

Table 2.1: Name of the conferences as starting venues for the literature review.

Field	Conference Name	Abbreviation
SE	International Conference on Automated Software Engineering	ASE
SE	International Conference on Fundamental Approaches to the Construction and Analysis of Systems	FASE
SE	International Conference on Performance Engineering	ICPE
SE	International Conference on Runtime Verification	RV
SE	International Conference on Software Engineering	ICSE
SE	International Conference on Software Maintenance and Evolution	ICSME
SE	International Conference on Tools and Algorithms for Software Engineering	TACAS
SE	International Symposium on Software Testing and Analysis	ISSTA
SE	International Symposium on the Foundations of Software Engineering	FSE
SE	Static Analysis Symposium	SAS
DB	International Conference on Data Engineering	ICDE
DB	International Conference on Management of Data	SIGMOD
DB	International Conference on Very Large Data Bases	VLDB

tool aims to give detection results during compilation, and can give suggested fixes. Most of the bug patterns encoded by Error Prone are general problems that are related to coding errors such as comparing arrays using "==". Infer (Facebook, 2016) is a static anti-pattern detection tool built by Facebook. The tool focuses on detecting problems in programming languages that are used for developing mobile apps (e.g., Java and Obj-C). Coverity (Coverity, 2016) is a commercial static anti-pattern detection tool, which looks for different kinds of bug patterns across various programming languages. Finally, AppScan (IBM, 2016a) is a static anti-pattern detection tool that is developed by IBM. The tool specializes in detecting security bugs. Although the above-mentioned static anti-pattern detection tools are widely used and are able to detect many kinds of bugs, none of them is focused on detecting functional or non-functional (i.e., performance) bugs related to database

Table 2.2: A list of popular static anti-pattern detection tools.

Tool	Focused bug types	Input	Language	Open sourced
FindBugs (Hovemeyer and Pugh, 2004)	General	Binary	Java	Yes
Error Prone (Google, 2016)	General	Binary	Java	Yes
Infer (Facebook, 2016)	General	Binary	Multiple	Yes
PMD (PMD, 2016)	General	Source	Multiple	Yes
Coverity (Coverity, 2016)	General	Source	Multiple	No
AppScan (IBM, 2016a)	Security	Source	Multiple	No

access code.

In terms of prior research, Dasgupta *et al.* (2009) propose a static analysis framework for database access code, and developers can use the framework to write detectors themselves. However, their framework only supports ADO.NET, and does not offer a list of pre-implemented patterns.

Finding 1. All surveyed state-of-the-art static anti-pattern detection tools are not geared towards detecting problems in database access code.

Nijjar and Bultan (2011) extract formal mathematical models from the database schema of Ruby on Rails applications, and look for errors in the models. In their follow-up work, Nijjar and Bultan (2013) extract formal mathematical models from the database schema, and develop heuristics to discover anti-patterns in the schema. Their framework can then automatically propose solutions to correct the database schema. Gould *et al.* (2004) propose a framework to validate SQL statically by analyzing the Java query strings. Gligoric and Majumdar (2013) propose a model checking approach to detect database concurrency issues in web applications, but their approach only found two problems in 12 studied applications. Zhang *et al.*

(2011) propose an approach that uses static analysis to discover violations of integrity constraints in database-centric applications based on four code anti-patterns. Since ORM queries are not statically typed, Cook and Rai (2005) present an approach for representing ORM queries as statically typed objects to avoid runtime errors.

There are some prior studies that aim to discover performance problems in database access code using program analysis. Smith and Williams (2003) first document the problem and possible solutions of a number of database-related performance anti-patterns. They discuss a pattern called *Empty Semi Trucks*, which occurs when a large number of excessive database query calls (e.g., select, insert, update, or delete) are sent to the database for a given task. However, Smith and Williams (2003) do not provide any detection algorithms for such problems. Hoekstra (2011) propose approaches to statically detect anti-patterns in database access code; however, the author did not provide any evaluation of the impact for the detected problems. A number of prior studies (Grechanik *et al.*, 2013a,b; Jula *et al.*, 2008) focus on detecting or preventing database deadlocks using program analysis. Pohjalainen and Taina (2008) propose an approach to analyze data usage in the application code, in order to automatically configure ORM frameworks. However, Pohjalainen and Taina (2008) do not provide the details of the algorithms nor do they implement nor evaluate their approaches. In summary, prior studies usually do not consider the efficiency of the database access calls (e.g., the performance of the SQLs generated by the ORM framework).

Chaudhuri *et al.* (2007), Tamayo *et al.* (2012), and Cao and Shasha (2013) propose approaches to link method calls to the corresponding generated SQLs. However, these approaches usually require developers to identify the problems themselves (i.e., similar to using profilers), instead of directly pinpointing developers to the problems.

Finding 2. Prior studies usually do not provide an automated evaluation or prioritization of the detected problems in the database access code, and cannot directly pinpoint developers to the root causes of the detected problems.

2.3 Code Transformation

Many prior studies in the database community use code transformation to improve the performance of database access code. There are two main directions of research on using code transformation to improve the quality of the database access code. One direction is to transform source code or byte code to improve the performance of SQL execution. Another direction is to transform and compile application code directly to SQL queries or stored procedures. Below, we discuss the approaches that are used by prior studies according to these two aforementioned directions.

Several prior studies improve the performance of database-centric applications by altering how the SQL queries are executed. Common approaches include batching (Chavan *et al.*, 2011a,b; Cheung *et al.*, 2014; Cook and Wiedermann, 2011; Guravannavar and Sudarshan, 2008; Iu and Zwaenepoel, 2006; Ramachandra *et al.*, 2015), pre-fetching (Ibrahim and Cook, 2006; Manjhi *et al.*, 2009; Ramachandra and Sudarshan, 2012; Ramachandra *et al.*, 2012), and asynchronous query execution (Chavan *et al.*, 2011a,b; Ramachandra *et al.*, 2015). These approaches improve

the performance of the database-centric application by reducing the round-trip time (RTT) between the application and the DBMS. However, these approaches do not consider how the queried (i.e., retrieved) data would be used in the application, since some of the queried data may not be needed in the code.

Cheung *et al.* (2012a,b, 2013b) propose an approach to separate the application and the database logic automatically, and transform the database logic to stored procedure calls. Stored procedure calls have better performance, since they are executed directly on the DBMS. Similarly, Wiedermann *et al.* (Wiedermann and Cook, 2007; Wiedermann *et al.*, 2008) propose an approach that uses static analysis to first understand how data will be used in the application code, and then transform the code logic to SQL. On the other hand, Cheung *et al.* (2013c) propose approaches to automatically transform Java code to SQL, and monitor the SQLs dynamically for load balancing. Cheung *et al.* (2013a) propose an approach to automatically synthesize SQL queries from Java code, based on pre- and post-conditions in Java methods. Cooper (2009) proposes an approach to automatically transform code written in functional programming languages to an equivalent SQL query. A potential problem of the above-mentioned approaches is that they add extra complexity to the application, which increases application maintenance difficulties. In addition, these approaches are rarely used in an industrial setting, since they do not involve domain experts (i.e., developers). Instead, these approaches remain as another layer of black boxes between the application code and the DBMS.

Finding 3. Many database access code transformation approaches only optimize database queries, and do not consider how the queried (i.e., retrieved) data would be used in the application code.

2.4 Domain specific languages and APIs

A number of prior studies propose domain-specific languages or APIs for database access code. Ackermann *et al.* (2015) propose a domain-specific query language for parallel execution of query code. Grust *et al.* (2009) propose a domain-specific language that can compile operations such as list iterations into SQL queries. Baltopoulos *et al.* (2011) propose a tool for SQL databases to allow transactions to be written in a functional language, and to be verified using an SMT-based refinement-type checker. Iu *et al.* (2010) propose a set of special Java APIs, which can be translated into complex SQL queries using bytecode transformation.

2.5 Chapter Summary

In this chapter, we survey related work that focuses on improving database access code. We find that existing static anti-pattern detection tools are not geared towards detecting problems in database access code. In addition, most prior studies do not prioritize the detected problems, and cannot pinpoint developers to the root cause. Finally, most prior studies focus on helping developers improve the performance of the SQL queries, but these studies do not consider how the queries data will be used in the application. In Chapter 5 and 6, we propose a static performance anti-pattern detection tool for database access code. In Chapter 7, we propose a dynamic anti-pattern detection approach that helps developers reduce redundant data accesses, by knowing what data is actually needed in the application. Finally, our proposed approaches also automatically rank the detected anti-patterns according to their performance impact.

CHAPTER 3

Background about Object-Relational Mapping

In this chapter, we provide some background knowledge of the ORM-based database access code, before introducing our research. Unlike SQL queries, which are the standard database access code for relational databases, ORM frameworks abstract SQL queries as a set of API calls and configurations. We first provide a brief overview of different ORM frameworks, then we discuss how database-centric applications access the DBMS using ORM frameworks. Our example is shown using the Java ORM standard, Java Persistence API (JPA), but the underlying concepts are common for other ORM frameworks.

3.1 Background on ORM Frameworks

ORM frameworks have become very popular among developers due to their convenience (Johnson, 2005; Leavitt, 2000). ORM frameworks support most modern programming languages, such as Java, C#, Ruby, and Python. Java, in particular, has a unified persistent API for ORM, called Java Persistent API (JPA). JPA has become an industry standard and is used in many open source and commercial applications (Sutherland and Clarke, 2016). Using JPA, users can switch between different ORM providers with minimal modifications.

There are many implementations of JPA, such as Hibernate (JBoss, 2016), OpenJPA (Apache, 2016), EclipseLink (Eclipse, 2016c), and parts of IBM WebSphere (IBM, 2016b). These JPA implementations all follow the Java standard, and share similar concepts and design. However, they may experience some implementation specific differences (e.g., varying performance (ObjectDB, 2016a)).

3.2 Translating Objects to SQL Queries

ORM frameworks are responsible for mapping and translating database entity objects to/from database records. Figure 3.1 illustrates such process in JPA. Although the implementation details and syntax may be different for other ORM frameworks, the fundamental idea is the same.

JPA allows developers to configure a class as a database entity class using annotations. There are three categories of annotations:

- **Entities and Columns:** A database entity class (marked as **@Entity** in the source code) is mapped to a database table (marked as **@Table** in the source

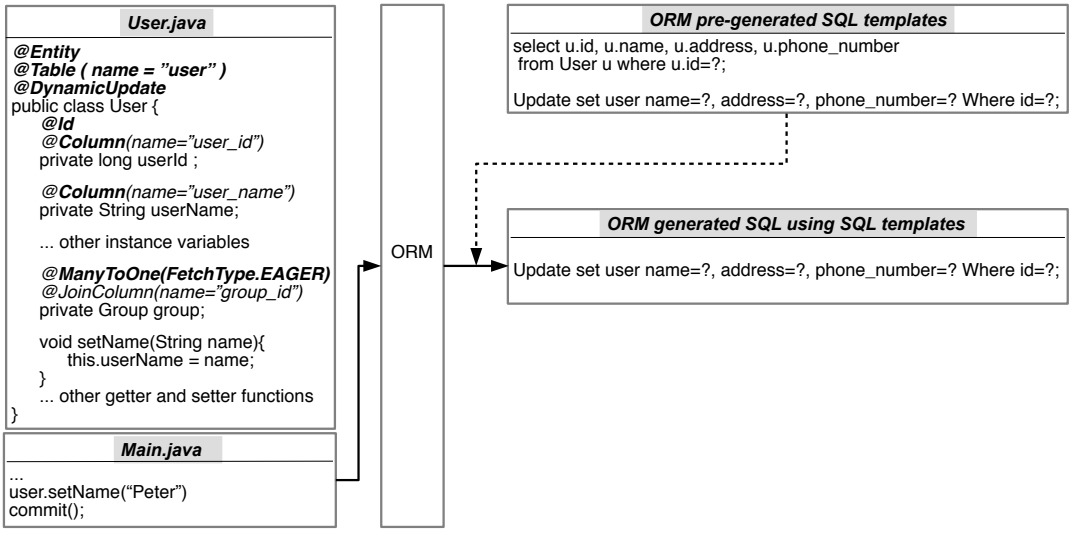


Figure 3.1: An example flow of how JPA translates object manipulation to SQL. Although the syntax and configurations may be different for other ORM frameworks, the fundamental idea is the same: developers need to specify the mapping between objects and database tables, the relationships between objects, and the data retrieval configuration (e.g., eager v.s. lazy).

code). Each database entity object is mapped to a record in the table. For example, the User class is mapped to the user table in Figure 3.1. **@Column** maps the instance variable to the corresponding column in the table. For example, the userName instance variable is mapped to the user_name column.

- **Relations:** There are four different types of class relationships in JPA: **OneToMany**, **OneToOne**, **ManyToOne**, and **ManyToMany**. For example, there is a **@ManyToOne** relationship between User and Group (i.e., each group can have multiple users).

- **Fetch Types:** The fetch type for the associated objects can be either ***EAGER*** or ***LAZY***. ***EAGER*** means that the associated objects (e.g., User) will be retrieved once the owner object (e.g., Group) is retrieved from the DBMS; ***LAZY*** means that the associated objects (e.g., User) will be retrieved from the DBMS only when the associated objects are needed (by the source code). Note that in some ORM frameworks, such as ActiveRecord (the default ORM for Ruby on Rails), the fetch type is set per each data retrieval, but other underlying principals are the same. However, most ORM frameworks allow developers to change the fetch type dynamically for different use cases (Eclipse, 2016d).

JPA generates and may cache SQL templates (depending on the implementation) for each database entity class. SQL templates are predefined strings, such as ”select u.id, u.name, u.address, u.phone_number from User u where u.id=?;“. JPA can generate complete SQL statements using the SQL templates by filling in the variable values (e.g., user id). The cached templates can avoid re-generating query templates to improve performance. These templates are used for retrieving or updating an object in the DBMS at run-time. As shown in Figure 3.1 (Main.java), a

developer changes the user object in the code in order to update a user's name in the DBMS. JPA uses the generated update template to generate the SQL queries for updating the user records. In this thesis, we focus on JPA due to its popularity, but our proposed approaches should be applicable to other ORM frameworks.

3.3 Caching

To optimize the performance and to reduce the number of calls to the DBMS, JPA, as well as most other ORM frameworks, uses a local memory cache (Keith and Stafford, 2008). When a database entity object (e.g., a User object) is retrieved from the DBMS, the object is first stored in the JPA cache. If the object is modified, JPA will push the update to the DBMS at the end of the transaction; if the object is not modified, the object will remain in cache until it is garbage collected or until the transaction is completed. By reducing the number of requests to the DBMS, the JPA cache reduces the overhead of network latency and the workload on database servers. Such caching mechanism provides significant performance improvement to applications that rely heavily on DBMSs.

Most caching frameworks act like an in-memory key-value store. When using JPA, these caching frameworks would store the database entity objects (objects that have corresponding records in the DBMS) in memory and assign each object a unique ID (i.e., the primary key). There are two types of caches in JPA:

- **Object cache.** As shown in workflow 1 (Figure 3.2), if the requested user object is not in the cache layer, the object will be fetched from the DBMS. Then, the user object will be stored in the cache layer and can be accessed as

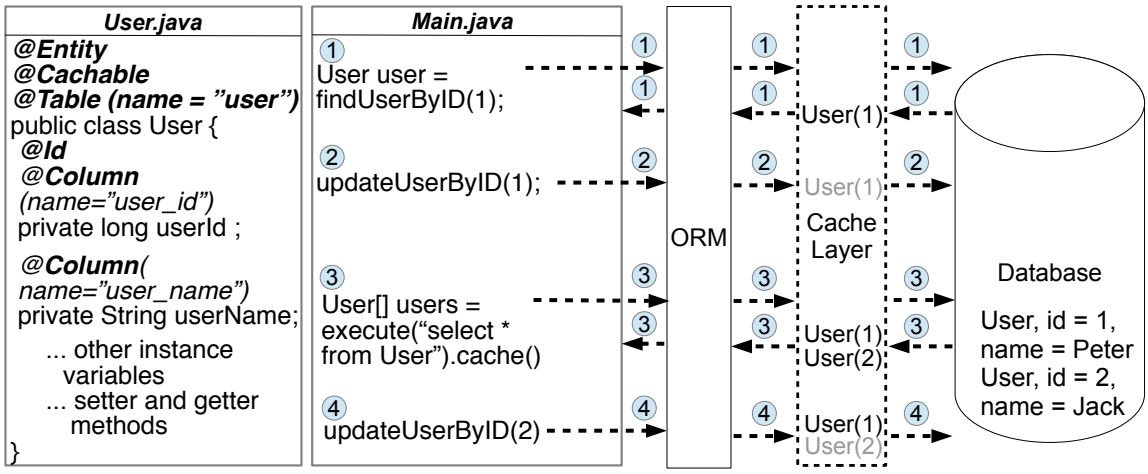


Figure 3.2: An example of simplified ORM code, ORM cache configuration code, and ORM cache mechanism. The numbers and the arrows indicate the flow for different workloads. The grey User objects in the cache layer means the objects are invalidated in the cache layer.

a key-value pair using its id (e.g., {*id: 1, User obj*}). If the object is updated, the cached data would be evicted to prevent a stale read (Workflow 2).

To cache database entity objects, developers must add an annotation *@Cachable* at the class declaration (as shown in User.java in Figure 3.2). Then, *all* database entity object retrieved by ID (e.g., retrieved using *findUserById()*) would be cached. These annotations configure the underlying caching frameworks.

- **Query cache.** The cache mechanism for query cache is slightly different from object cache. For example, the cached data for a *select all* query on the user table (Workflow 3) would look as follows:

$$\left\{ \begin{array}{l} \textit{select * from User} \rightarrow \{id:1, id:2\} \\ \{id:1, id:2\} \rightarrow \{id:1, \text{User obj}\}, \{id:2, \text{User obj}\} \end{array} \right\}$$

The cache layer stores the *ids* of the objects (i.e., id 1 and 2) that are retrieved by the query, and uses the ids to find the cached objects (the corresponding

Table 3.1: Pros and cons of object cache.

Pros	Cons
Lower cache miss cost Easier to determine where to add cache	Lower performance improvement if cache hit

Table 3.2: Pros and cons of query cache.

Pros	Cons
Higher performance improvement if cache hit	High cache miss cost Hard to determine whether cache should be added

User obj). Thus, the object cache must be enabled to use a query cache. When a user object is updated (workflow 4), the query cache needs to retrieve the updated object from the DBMS to prevent a stale read. Thus, if the queried entity objects are frequently modified, using a query cache may not be beneficial, and may even hinder performance (Ferreira, 2016).

To cache query results, developers must call a method like cache() before executing the query (Main.java in Figure 3.2). Such method is used to configure the underlying caching frameworks.

Table 3.1 and 3.2 summarize the pros and cos of object and query cache.

Adding caches incorrectly can introduce overhead to the application. Caching a frequently modified object or query will cause the caching framework to constantly evict and renew the cache, which not only cause cache renewal overhead but may also result in executing extra SQL queries (Linwood and Minter, 2010). Therefore, blindly adding caches without understanding the workload can cause performance degradation (Smith, 1985).

3.4 Chapter Summary

In this chapter, we provide some background knowledge of the ORM-based database access code. In this thesis, we focus on improving the performance of such database access code due to the popularity of ORM frameworks. In the later chapters of the thesis, we propose approaches to help developers detect problems in their ORM-based database access code and automatically find optimal ORM cache configurations.

CHAPTER 4

Maintenance Activities of ORM Code

Despite the advantages of using ORM frameworks, we observe several difficulties in maintaining ORM code when cooperating with our industrial partner. After conducting studies on other open source applications, we find that such difficulties are common in other applications. Our study finds that i) ORM cannot completely encapsulate database accesses in objects or abstract the underlying database technology, thus may cause ORM code changes more scattered; ii) ORM code changes are more frequent than regular code, but there is a lack of tools that help developers verify ORM code at compilation time; iii) we find that changes to ORM code are more commonly due to performance or security reasons; however, traditional static code analyzers need to be extended to capture the peculiarities of ORM code in order to detect such problems. Our study highlights the hidden maintenance costs of using ORM frameworks and provides some initial insights about potential approaches to help maintain ORM code.

An earlier version of this chapter is published at the 13th International Conference on Mining Software Repositories (MSR), 2016. Austin, Texas. Pages 165–176. (Chen *et al.*, 2016c)

4.1 Introduction

MANAGING data consistency between source code and database management systems (DBMSs) is a difficult task, especially for complex large-scale applications. As more applications become heavily dependent on DBMSs, it is important to abstract the database accesses from developers. Hence, developers nowadays commonly make use of Object-Relation Mapping (ORM) frameworks to provide a conceptual abstraction between objects in Object-Oriented Languages and data records in the underlying DBMS. Using ORM frameworks, changes to object states are automatically propagated to the corresponding database records. However, there may also be costs associated with maintaining ORM code. Therefore, in this chapter, we aim to study the maintenance activities that are associated with ORM code, and understand how to help developers maintain ORM code.

A recent survey (ZeroturnAround, 2014) shows that 67.5% of Java developers use ORM frameworks (i.e., Hibernate) to access the database, instead of using Java Database Connectivity (JDBC) or other frameworks. However, despite ORM's popularity and simplicity, maintaining ORM code (i.e., code that makes use of ORM frameworks) may be very different from maintaining regular code, since ORM code abstracts the DBMS accesses. Prior studies (Curino *et al.*, 2008a; Gobert *et al.*, 2013; Qiu *et al.*, 2013) usually focus on the evolution and maintenance of database schemas. However, the maintenance of database access code, such as ORM code, is rarely studied. In particular, since ORM introduces another abstraction layer on top of SQL, introducing extra burden for developers to understand the exact behaviour of ORM code (Chen *et al.*, 2014a), maintaining ORM code can be effort consuming.

During a recent cooperation with one of our industrial partners, we examined the challenges associated with maintaining a large-scale Java software application that uses ORM frameworks to abstract database accesses. We observed several difficulties in maintaining ORM code in Java applications. For example, changes that involve ORM code are often scattered across many components of the application (modifies many times more files than that of regular code). We conjecture that such scatterness may be caused by ORM's inability to completely abstract database accesses.

With such observations on the industrial application, we sought to study several open source Java applications that heavily depend on ORM in order to verify whether maintaining ORM code in these applications also suffers from the difficulties that are observed in the industrial application. We conducted an empirical study on three open source Java applications, in addition to the one large-scale industrial Java application. We find that the difficulties in maintaining ORM code are common among the studied applications, which further highlights that the challenges of maintaining ORM code is a wide ranging concern.

In particular, we investigate the difficulties of maintaining ORM code through exploring the following research questions:

RQ1: ***How Localized are ORM Code Changes?***

We find that code changes that involve ORM code are more scattered and complex, *even after we control for fan-in* (statistically significant). In other words, ORM cannot completely encapsulate the underlying database accesses in objects, which may make ORM code harder to maintain compared to regular code.

RQ2: ***How does ORM Code Change?***

We find that ORM code is changed more frequently (15%–79% more) than non-ORM code. In particular, ORM model and query code are often changed, which may increase potential maintenance problems due to lack of query return type checking at compilation time (hence many problems might remain undetected till runtime in the field). On the other hand, developers do not often tune ORM configurations for better performance. Therefore, developers may benefit from tools that can automatically verify ORM code changes or tune ORM configurations.

RQ3: ***Why does ORM Code Change?***

Through a manual analysis of ORM code changes, we find that compared to regular code, ORM code is more likely changed due to performance and security reasons. However, since ORM code is syntactically (i.e., have a unique set of APIs, and different code structure/grammar) and semantically (i.e., access databases) different from regular code, traditional static code analyzers need to be extended to capture the peculiarities of ORM code in order to detect such problems.

4.2 The Main Contributions of this Chapter

Our study highlights that practitioners need to be aware of the maintenance cost when taking advantage of the conveniences of ORM frameworks. Our study also provides some initial insights about potential approaches to help reduce the maintenance effort of ORM code. In particular, our results helped our industrial partner understand that some types of problems may be more common in ORM code, and what kinds of specialized tools may be needed to reduce the maintenance effort of ORM code.

4.3 Related Work

In this chapter, we study the characteristics and maintenance of ORM code, which is not yet studied nor well understood by researchers and practitioners. In this section, we survey prior studies on the evolution of database code and non-code artifacts. While many prior studies examined the evolution of source code, (e.g., Gall *et al.* (1997); Godfrey and Tu (2000); Lehman *et al.* (1997)), this chapter studies the evolution of software applications from the perspective of the non-code artifacts. Such non-code artifacts are extensively used in practice, yet the relation between such artifacts and their associated source code is not widely studied.

4.3.1 Evolution of database code

Prior studies focus on the evolution of database schemas, while our chapter focuses on the evolution of the database-related code. Qiu *et al.* (2013) conduct a large-scale study on the evolution of database schema in database-centric applications. They study the co-evolution between database schema and application code, and they find that database schemas evolve frequently, and cause significant code-level modifications (we observe similar co-evolution even though ORM is supposedly designed to mitigate the need for such co-evolution). Curino *et al.* (2008b) study the database schema evolution on Wikipedia, and study the effect of schema evolution on the application front-end. Curino *et al.* (2008a) design a tool to evaluate the effect of schema changes on SQL query optimization. Meurice and Cleve (2014) and Gobert *et al.* (2013) develop a tool for visualizing database schema evolution. Goeminne *et al.* (2014) analyze the co-evolution between code-related and

database-related activities in a large open source application, and found that there was a migration from using SQL to ORM. In another work, Goeminne and Mens (2015) study the survival rate of several database frameworks in Java projects (i.e., when would a database framework be removed or replaced by other frameworks), and they found that JPA has a higher survival rate than JDBC.

4.3.2 Non-code artifacts

User-visible features. Instead of studying the code directly, some studies have picked specific features and followed their implementation throughout the lifetime of the software application. Anton and Potts (2003) study the evolution of telephony software applications by studying the user-visible services and telephony features in the phone books of the city of Atlanta. They find that functional features are introduced in discrete bursts during the evolution. Kothari *et al.* (2008) propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated during the execution of these features. Greevy *et al.* (2006) use program slicing to study the evolution of features. His and Potts (2000) study the evolution of Microsoft Word by looking at changes to its menu structure. Hou and Wang (2009) study the evolution of UI features in the Eclipse IDE.

Communicated information. Shang *et al.* (2011, 2014) study the evolution of communicated information (*CI*) (e.g., log lines). They find that *CI* increases by 1.5-2.8 times as the application evolves. Code comments, which are a valuable instrument to communicate the intent of the code to programmers and maintainers, are another source of *CI*. Jiang and Hassan (2006) study the evolution of source

code comments and discover that the percentage of functions with header and non-header comments remains consistent throughout the evolution. Fluri *et al.* (2007, 2009) study the evolution of code comments in eight software applications. Hassan and Holt (2006) propose an approach to recover co-change information from source control repositories, and Malik *et al.* (2008) study the rational for updating comments. Ibrahim *et al.* (2012) study the relationship between comments and bugs in software applications.

4.4 Preliminary Study

In this section, we first introduce our studied applications, then we discuss the evolution of ORM code in these applications.

4.4.1 Studied Applications

We study three open-source applications (Broadleaf Commerce (Commerce, 2013), Devproof Portal (Portal, 2015), and JeeSite (ThinkGem, 2016)) and one large-scale industrial application (EA) by mining their git repositories. Due to a Non-Disclosure Agreement (NDA), we cannot expose all the details of EA. Table 4.1 shows an overview of the studied applications, as well as the overall ORM code density. All of the studied applications follow the typical Model-View-Controller (MVC) design (Krasner and Pope, 1988), and use Hibernate as the implementation of ORM. Broadleaf Commerce is an e-commerce application, which is widely used in both open-source and commercial settings. Devproof Portal is a fully featured

Table 4.1: Statistics of the studied applications in the latest version. EA is not shown in detail due to NDA.

	Lines of code (K)	No. of Java files	% files contain ORM code	No. of studied versions	Latest studied version	Median ORM code density among all all versions
Broadleaf	363K	2,249	13%	79	3.1.0	1.3%
Devproof	53.7K	541	20%	7	1.1.1	0.7%
JeeSite	397K	126	16%	5	1.0.4	1.1%
EA	>300K	>3,000	4%	>10	—	< 1%

portal, which provides features such as blogging, article writing, and bookmarking. JeeSite provides a framework for managing enterprise information, and offers a Content Management System (CMS) and administration platform. EA is a real-world industrial application that is used by millions of users worldwide on a daily basis.

4.4.2 Evolution of ORM Code

We first conduct a preliminary study on the evolution of ORM code. We use the following metrics to study the evolution of ORM code:

- Number of database table mappings;
- Number of ORM query calls;
- Number of performance configurations (e.g., caching or batching);
- ORM code density.

We define ORM code density as the total number of ORM code (i.e., lines of code that perform database table mapping, performance configuration calls, and ORM

query calls) divided by the total lines of code. Below we discuss our findings for the above-mentioned metrics.

Figure 4.1 shows the evolution of the three types of ORM code. We find that, in general, the number of ORM query calls has the steepest increase overtime. We also find that the number of DBMS table-mappings does not change much overtime, and that the total number of ORM performance configurations remains relatively stable in JeeSite and EA. We also find that the change rate of ORM configuration code is lower than the other types of ORM code in some applications. Since ORM configuration code is usually applied on ORM queries and ORM table mapping code, this finding may indicate that not all developers spend enough time tuning or adding the configurations, which may result in performance problems in ORM-based applications (Chen *et al.*, 2014a, 2016a,d).

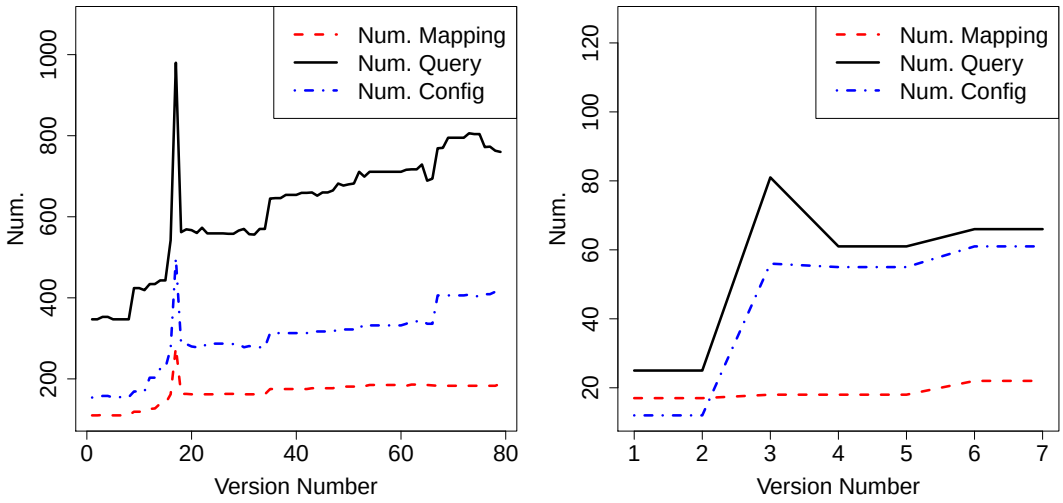
4.5 Case Study Results

We now present the results of our research questions. Each research question is composed of four parts: motivation, approach, experimental results, and discussion.

RQ1: How Localized are ORM Code Changes?

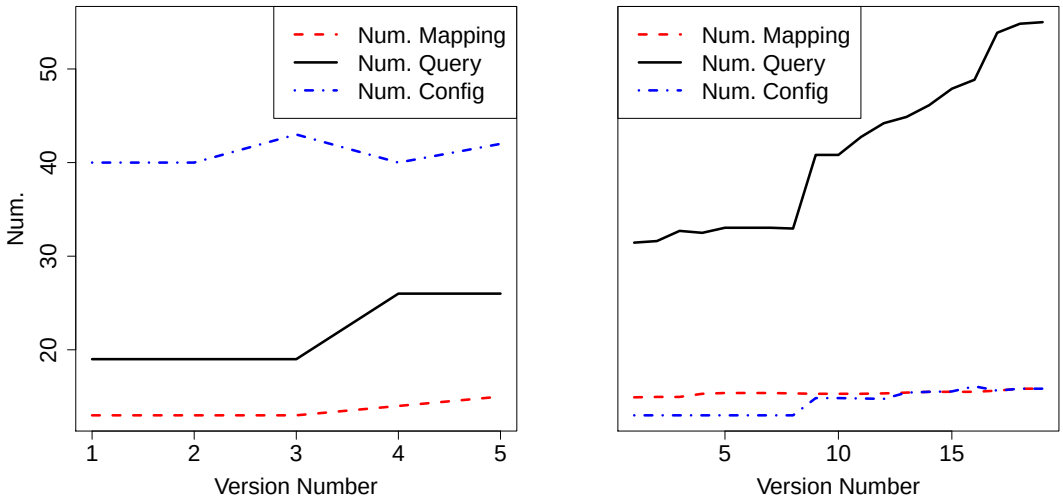
Motivation. To verify the generalizability of our observation in EA and examine if ORM code changes are more scattered (i.e., complex) by nature, we study the complexity of ORM code changes in this RQ.

Approach. We study the code change complexity after normalizing the fan-in of ORM code. High degree of dependence (i.e., high fan-in) is likely to lead to higher



(a) Broadleaf

(b) Devproof



(c) JeeSite

(d) EA

Figure 4.1: Evolution of the total number of database table mappings, ORM query calls, and ORM configurations. The values on the y-axis are not shown for EA due to NDA.

maintenance costs (due to changing more dependent files). Hence, by controlling for fan-in, we ensure that our observations are more likely due to the nature of ORM code.

Dependence on the Files that Contain ORM Code. To measure the dependence on the files that contain ORM code, we compute the degree of fan-in at the file level (Henry and Kafura, 1981). We annotate each file as one that contains ORM code (ORM file) or one that does not contain ORM code (non-ORM file). Fan-in measures the number of files that depend on a given file. For example, if file B and C both are calling one or more methods in file A, then the degree of fan-in of file A is two. We compute the fan-in metric for all files for all studied versions, and then we compare the degree of fan-in between ORM and non-ORM files.

Complexity of ORM Code Change. To measure the complexity of ORM code changes, we compute the following metrics for each commit:

- Total number of code churn (lines inserted/deleted) in a commit;
- Total number of files that are modified in a commit;
- Commit change entropy (Hassan, 2009).

The three above-mentioned metrics are used in prior studies to approximate the complexity of code changes in a commit (Chen *et al.*, 2014b; Soh *et al.*, 2013; Zaman *et al.*, 2011). We control for fan-in in our study by dividing each metric by the value of fan-in in the associated files. The reason is that each metric may be correlated with fan-in to a certain degree, and a division allows us to minimize the effect of such correlations. We classify commits into ORM commits (i.e., commits that modify ORM code), and non-ORM commits, and compare the metrics between

these two types of commits.

Entropy measures the uncertainty in a random variable, and maximum entropy is achieved when all the files in a commit have the same number of modified lines. In contrast, minimum entropy is achieved when only one file has the total number of modified lines in a commit. Therefore, higher entropy values represent a more complex change (i.e., scattered changes), where smaller entropy values represent a less complex change (i.e., changes are concentrated in a small number of files) (Hassan, 2009).

To measure the change entropy, we implement the normalized Shannon Entropy to measure the complexity of commits (Hassan, 2009; Zaman *et al.*, 2011). The entropy is defined as:

$$H(Commit) = \frac{-\sum_{i=1}^{n} p(File_i) * log_e p(File_i)}{log_e(n)}, \tag{4.1}$$

where n represents the total number of files in a commit $Commit$, and $H(Commit)$ is the entropy value of the commit. $p(File_i)$ is defined as the number of lines changed in $File_i$ over the total number of lines changed in every file of that commit. For example, if we modify three files A (modify 1 line), B (modify 1 line), and C (modify 3 lines), then p(A) will be $\frac{1}{5}$ (i.e., $\frac{1}{1+1+3}$).

Statistical Tests for Metrics Comparison. To compare the metric values between ORM and non-ORM files, we use the single-sided Wilcoxon rank-sum test (also called Mann-Whitney U test). We choose the Wilcoxon rank-sum test over Student's t-test because our metrics are skewed, and the Wilcoxon rank-sum test is a non-parametric test, which does not put any assumption on the distribution of two populations. The Wilcoxon rank-sum test gives a p-value as the test outcome. A p-value ≤ 0.05 means that the result is statistically significant, and we may reject the

null hypothesis (i.e., the two populations are different). By rejecting the null hypothesis, we can accept the alternative hypothesis, which tells us if one population is statistically significantly larger than the other. In this RQ, we set the alternative hypothesis to check whether the metrics for ORM commits are *larger* than that of non-ORM commits. Prior studies have shown that reporting only the p-value may lead to inaccurate interpretation of the difference between two populations (Kampenes *et al.*, 2007; Nakagawa and Cuthill, 2007). When the size of the populations is large, the p-value will be significant even if the difference is very small. Thus, we report the effect size (i.e., how large the difference is) using *Cliff's Delta*, which is a non-parametric effect size measure that does not have any assumption on the distribution of the population (Cliff, 1993). The strength of the effects and the corresponding range of *Cliff's Delta* values are (Romano *et al.*, 2006):

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } \textit{Cliff's Delta} < 0.147 \\ \text{small} & \text{if } 0.147 \leq \textit{Cliff's Delta} < 0.33 \\ \text{medium} & \text{if } 0.33 \leq \textit{Cliff's Delta} < 0.474 \\ \text{large} & \text{otherwise} \end{cases}$$

Results. ***We find that files that contain ORM code have a higher degree of fan-in (statistically significant) in the studied applications, which make them good candidate applications for our study.*** We compute the degree of fan-in for each version separately, and report the p-value of comparing the degrees of dependency from ORM and non-ORM code of each version. The p-value is statistically significant (<0.05) in every version of all the studied applications (ORM files have a higher fan-in). Table 4.2 shows the median of the effect sizes across all versions of the degree of fan-in of files that contain ORM code and files that do not contain ORM code.

The effect sizes of the difference are non-trivial in all of our studied applications. These findings highlight and confirm the central role of ORM code in the studied software applications and their evolution. Thus, these applications are indeed good candidates for our study.

ORM-related commits modify more lines of code and files than other types of commits, and the commits are more scattered, even after we control for fan-in. We obtain a p-value of $<< 0.001$ for the result of the Wilcoxon rank-sum test for the total code churn, the total files modified, and the change entropy of a commit in the studied applications. Our findings indicate that commits that modify ORM code are more complex (statistically significant) than commits that do not modify ORM code. From Table 4.3 we can also see that the median of the metrics for commits that modify ORM code are all larger than commits that do not modify ORM code. We find that 88% of the median effect sizes of the ORM code complexity is at least medium, which further supports the result of our Wilcoxon rank-sum test.

Since we find that files with ORM code have a higher fan-in in general, such characteristics may affect the complexity of changes that involve ORM code. As a result, we further study the complexity of ORM code changes after controlling for fan-in. We calculate the total fan-in of all the files in each commit, and then we normalize the commit complexity by dividing it by the total fan-in of all involved files. For example, if a commit modifies 1,000 lines of code, and the total fan-in of all the files that are modified in the commit is 100, the normalized total lines of code modified is 10 (1,000/100). We find that, after controlling for fan-in, commits that modify ORM code are still more complex (all statistically significant, except for change entropy in JeeSite).

Table 4.2: Medians (*Med.*, computed across all versions) and effect sizes (*Eff.*, median across all versions) of fan-in of ORM and non-ORM files. All differences are statistically significant. We only show the effect sizes for EA due to NDA.

Metric	Type	Broadleaf		Devproof		JeeSite		EA
		Med.	Eff.	Med.	Eff.	Med.	Eff.	Eff.
Fan-in	ORM	11.0	0.29	5.9	0.32	6.2	0.81	0.34
	Non-ORM	6.8		3		2		

Table 4.3: Medians (*Med.*, computed across all versions) and effect sizes (*Eff.*, averaged across all versions) of the complexity of ORM code changes. All differences are statistically significant. We only show the effect sizes for EA due to NDA.

Metric	Type	Broadleaf		Devproof		JeeSite		EA
		Med.	Eff.	Med.	Eff.	Med.	Eff.	Eff.
LOC modified	ORM	102	0.44	241	0.60	773	0.76	0.50
	Non-ORM	17		30		21		
Files modified	ORM	6	0.53	11	0.51	13	0.69	0.53
	Non-ORM	2		3		2		
Entropy	ORM	0.78	0.34	0.80	0.29	0.85	0.37	0.28
	Non-ORM	0.00		0.59		0.61		

It may first seem expected that, since ORM code is one of the core components of an application (i.e., has higher fan-in), ORM code changes should be more scattered. However, the finding highlights a potential issue with ORM code. Although the goal of ORM code is primarily on abstracting relational databases in object-oriented programming languages, we find that ***the underlying database access is not completely encapsulated inside objects, so changing ORM code requires changing many other files*** (we conduct a manual study in RQ3 to understand the reasons for changing ORM code). Further studies are needed to better understand and resolve the design problem of ORM code in order to keep database knowledge completely encapsulated within objects.

Table 4.4: Number of ORM files in the top 100 files with the largest degree of fan-in (averaged across versions).

Metric	Broadleaf	Devproof	JeeSite	EA
# of files with ORM code in the top 100	11.5	35	18	14.3

Table 4.5: Percentage of files that contain each type of ORM code in the top 100 high fan-in files.

Type	Broadleaf	Devproof	JeeSite	EA
Data Model	7%	8%	17%	16%
ORM Query Call	1%	26%	3%	1%
Perf. Config. Call	3%	8%	15%	10%

Discussion. We do not know whether the high fan-in of ORM files is caused by the design of ORM code, or simply because these files are the core components of the studied applications. Thus, we conduct an experiment on the degree of fan-in of files with/without ORM code. We first find the top 100 files with the highest fan-in values for each studied application, and examine how many of the top 100 files have ORM code (Table 4.4). We find that in all the studied applications, about 11–35 files among the top 100 files contain ORM code. In short, we find that files with ORM code are not the only core components of an application; nevertheless files with ORM code tend to have a higher change complexity.

Our finding helps our industrial partner recognize the high scatteredness of ORM code changes. ORM code changes may be much riskier, and require more careful attention and reviewing. Our findings are consistent among the studied open source applications, and the problems are not specific to EA. Further studies are needed to understand the reasons that ORM code cannot completely encapsulate the underlying database concerns within objects.

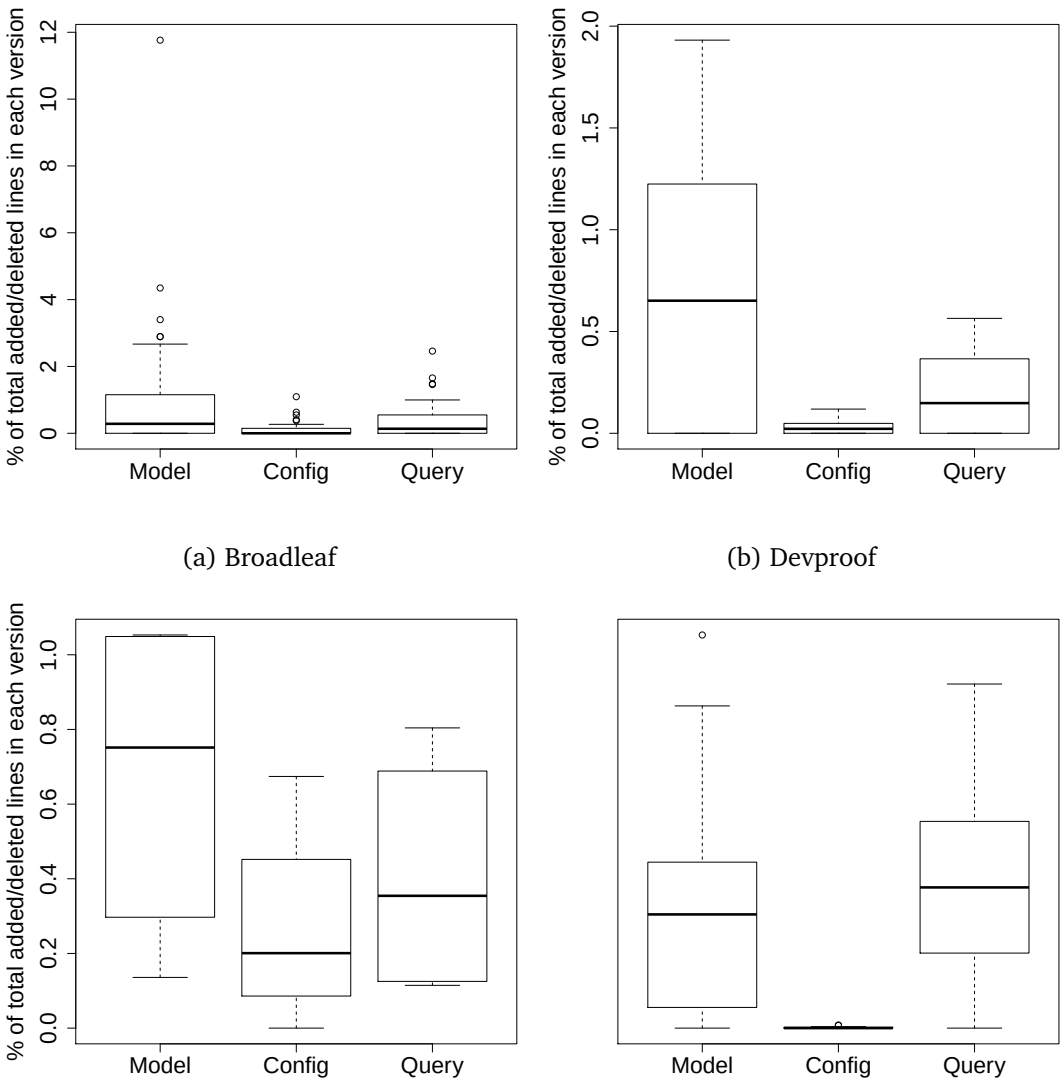
We find that ORM cannot completely encapsulate the underlying database accesses in objects, which may be the reason for the scattered nature of ORM code changes.

RQ2: How does ORM Code Change?

Motivation. In RQ1, we find that changes involving ORM code are usually more complex and more scattered. In this RQ, we want to further study the change frequency of each type of ORM code to find out which type of ORM code requires the most maintenance effort. Namely, we study how developers change different types of ORM code (i.e., data model, ORM query, and performance configuration).
Approach. To answer this RQ, we compute the total code churn (i.e., added and deleted lines) and ORM code churn (i.e., added and deleted ORM code) between versions. We are particularly interested in how developers change different types of ORM code, since these changes directly reflect the maintenance activity on ORM code. Therefore, we compute the following metrics:

- Total code churn;
- Total ORM code churn;
- Code churn for ORM data model, query calls, and performance configurations.

ORM data model code churn ($churn_{\text{model}}$) computes the amount of churn related to data model code. ORM query-related code churn ($churn_{\text{query}}$) measures the amount of ORM query-related churns. ORM performance configuration code churn ($churn_{\text{config}}$) measures the amount of ORM performance configurations churns (e.g.,



(a) Broadleaf

(b) Devproof

(c) JeeSite

(d) EA

Figure 4.2: Distribution of the percentage of code churn for different types of ORM code changes across all the studied versions. We omit the scale for EA due to NDA.

Table 4.6: Total code churn and ORM-related code churn in the studied applications.

	Code Churn	ORM Churn	$churn_{\text{model}}$	$churn_{\text{config}}$	$churn_{\text{query}}$
Broadleaf	1472K	22K (2%)	67%	7%	26%
Devproof	88K	1.1K (1%)	68%	5%	27%
JeeSite	11K	194 (2%)	49%	18%	33%
EA	—	<1%	41%	0.2%	59%

cache configurations). We also measure the total amount of code churn and ORM code churn for comparison.

We use the following metric to measure the churn ratio of ORM code:

$$churn_{\text{ratio}} = \frac{\text{ORM code churn}}{\text{ORM code density}}, \tag{4.2}$$

where a high $churn_{\text{ratio}}$ value means that ORM code has a higher chance of being modified (assuming each line of code has the same probability of being modified).

Results. ***Tools such as type checker for ORM queries and automated configuration tuning may help developers with ORM code maintenance.*** In Broadleaf, we find that about 2% (22K LOC) of total code churn (across all versions) is related to ORM (Table 4.6). We also find similar amount of ORM code churn in Devproof (1% of all churn) and JeeSite (2% of all churn). Given the low ORM code density in the entire application (Table 4.1), this implies that ORM code changes more frequently than other code. The $churn_{\text{ratio}}$ for Broadleaf, Devproof, JeeSite, and EA are 115%, 179%, 164%, and 120%, respectively.

In Broadleaf, Devproof, and JeeSite, most ORM code changes are related to data models. Developers sometimes make changes to the data models about how the classes are mapped to the database tables. Such changes may lead to other ORM code changes. Consider the following example from the studied applications:

```
1 -@ManyToMany(fetch = FetchType.LAZY,
2 -targetEntity = OrderItemImpl.class)
3 -@JoinTable(name = "GIFTWRAP_ORDERITEM")
4 +@OneToMany(fetch = FetchType.LAZY,
5 +mappedBy = "giftWrapOrderItem",
6 +targetEntity = OrderItemImpl.class)
7 private List<OrderItem> wrappedItems =
8 new ArrayList<OrderItem>();
```

Developers change the relationship between OrderItemImpl and GitWrapOrderItem from **@ManyToMany** to **@OneToMany** due to data model changes and refactoring. Such changes may cause some other side effects such as the properties of the data model in the code (e.g., entity relationships) no longer matching the properties in the DBMS (Nijjar and Bultan, 2013).

We find that developers in all the studied applications also change ORM query calls very often. These ORM query calls provide developers a non-encapsulated (i.e., not Object-Oriented) way to retrieve data from the database. However, evolution of database models and frequent changes to these ORM query calls may cause some problems, since there exists no type checking for ORM query calls at compilation time (Bauer and King, 2005). For example, Nijjar and Bultan (2013) found that there may exist problems in ORM data models due to the abstraction of the underlying relational models. ***Therefore, having tools that can help developers with compile time code verification or type checking for ORM query calls may reduce ORM code maintenance effort***.

Finally, we find that changes to ORM performance configurations are less frequent than the other types of ORM code. This finding is alarming, since the performance of ORM code is related to how ORM code is configured (Chen *et al.*, 2014a, 2016d). Prior studies (Dageville *et al.*, 2004; Yagoub *et al.*, 2008) from the database community have shown that tuning the performance of database-related code (e.g., SQL) is a continuous process, and needs to be done as applications evolve. ***As a result, automatically helping developers configure ORM code will be of great value when maintaining ORM code***. In Chapter 8, we follow up on this finding, and implement an automated tool for tuning ORM performance configuration code. We find that our tool helps improve application throughput by 27–138%.

Discussion. Since the amount of each type of ORM code is different, we normalize the churn by the total existence of each type of ORM code in the application. For each type of ORM code, we divide the number of modified lines of code by the total amount of such code. We compute such number for each version, and we report the median value in Table 4.7. We find that the change size of ORM query code is about 1.5%–56.8% of all ORM query code, which is significantly larger than the other two types of ORM code. We find a consistent trend in all studied applications. Our result shows that ORM query code is changed more often (after normalization), even though ORM query code bypass the ORM abstraction layer and may be error-prone (no type checking at compile time).

Table 4.7: Median churn percentage of each type of ORM code across all versions.

	Model	Config	Query
Broadleaf	0.9%	0.0%	1.5%
Devproof	45.6%	9.8%	56.8%
JeeSite	21.8%	17.8%	38.0%
EA	5.4%	0.0%	8.9%

ORM code has less code density but is changed more frequently (15%–79% more) than non-ORM code. We also find that both ORM query calls and models are changed frequently, while ORM configurations are rarely changed. Developers may benefit from tools for verifying the type of returned objects by ORM queries, or tools for automated tuning of ORM performance configurations.

RQ3: Why does ORM Code Change?

Motivation. In the previous RQs, we study the characteristics of different ORM code changes. However, the reasons for changing the ORM code are not known. Since ORM code is dependent on the DBMS, the reasons for ORM code changes may be different from regular code changes. However, ORM code may also be different from regular SQL queries, since ORM abstracts SQL queries from developers. Thus, in this RQ, we manually study the reasons for ORM code changes, and we compare such reasons with regular code changes (i.e., changes that do not modify ORM code).

Approach. We manually study the reasons for the changes that developers make. We first collect all the commits, and we annotate each commit as ORM-related or regular commits (commits that do not modify ORM code). In total, there are 2,223 commits that change ORM code, and 9,637 commits that do not change ORM

Table 4.8: Manually derived categories for commits.

Category	Description	Abbr.
Bug Fix	Code is modified to fix a bug	Bug
Compatibility Issue	Modifications to allow code to work in an other environment	Compat
Feature Enhancement	Enhance current functionalities	Enhance
New Feature	Add new functionalities	New
Performance Config.	Performance and performance configuration tuning	Config
Refactoring	Code refactoring	Refactor
Model	Database schema/ORM data model is changed	Model
Security Enhancement	Enhance security	Secure
Test	Add test code	Test
Upgrade	Upgrade dependent changes	Upgrade
GUI	Modified graphical/web user interface	GUI
Documentation	Updated documentation	Doc
Build	Modify/Update build files	Build

code (across all the studied open source applications). We do not show data from EA in this study due to NDA, but our findings in EA (e.g., the categories and the distributions) are very similar to what we found in the open source applications. In order to achieve a confidence level of 95% with a confidence interval of 5% in our results (Boslaugh and Watters, 2008), we randomly sample 328 ORM-related commits and 369 non-ORM commits for our manual study. We first examine the randomly sampled commits (both ORM and non-ORM commits) with no particular categories in mind. Then, we manually derive the set of categories for which these commits belong. In total, we derive 13 categories for the commits. Table 4.8 has the descriptions for the categories. For commits that belong to multiple categories, we assign the commits to all the categories to which they belong. We then study how the studied commits are distributed in these categories.

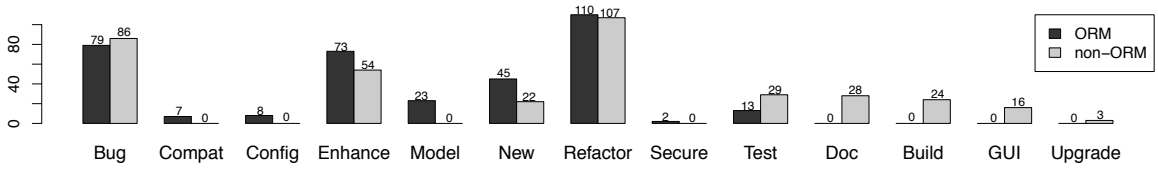


Figure 4.3: Distributions of the commits for ORM and non-ORM code in each categories.

Results. ***ORM code changes are more likely due to performance, compatibility, and security problems compared to regular code; automated techniques for detecting such problems in ORM-based applications may be beneficial.*** Figure 4.3 shows the distributions of the studied commits and the category to which they belong. We find that ORM and non-ORM code changes share some common reasons. Developers spend a large amount of effort on ORM code refactoring (30.64% of all commits that modify ORM code). In addition, 22.01% of the commits are related to bug fixing, and 20.33% of the commits are related to feature enhancement. In short, more than 70% of the commits that change ORM code are related to code maintenance activities (i.e., refactoring, enhancement, and bug fixing) (Herraiz *et al.*, 2013). We find that these maintenance activities have very similar distributions in non-ORM commits.

Nevertheless, some categories of ORM commits do not exist in non-ORM commits (i.e., *Compat*, *Config*, *Model*, and *Secure*). Our sampled commits show that these reasons are more likely to result in ORM code changes, although we expect that *Model* only exists in commits that contain ORM code. We also find that ORM frameworks play a central role in the performance of ORM-based applications – developers change ORM configuration code more frequently for performance improvement. In RQ2 we find that ORM configuration code is not frequently changed.

However, a prior study (Chen *et al.*, 2014a) finds that developers may not always be aware of the performance impact of the ORM code due to the database abstraction (i.e., developers may not know the code they write would result in slow database accesses). Hence, there may be many places in the code that require performance tuning, and tools that can automatically change/tune ORM configuration code can be beneficial to developers.

Moreover, even though ORM ideally should ensure that the code is database-independent (i.e., porting an application to a different database technology should require no code changes), we still find some counterexamples (see our discussion below). Finally, since ORM code need to send user requests to the underlying DBMSs, security may also more likely to be a concern (e.g., SQL injection attacks). In the discussion, we further discuss some of the compatibility, performance, and security problems that we found in our manual study.

Discussion. Even though ORM is touted as a solution that would ensure that ORM code would work against all kinds of database technologies, we still observe DBMS compatibility issues. For example, in Broadleaf, developers change the database table name due to an Oracle-specific size limit on table names.

During our manual study, the most commonly observed ORM performance configuration changes are due to data caching. Consider the following example from Broadleaf:

```
1 +@Cache(usage =
2 +CacheConcurrencyStrategy.READ_ONLY)
3 private Map<String, String> images;
```

The images variable stores a binary image for each category of items (represented

using a Java String). The image for each category does not change often, yet they often have a large data size. Frequently retrieving the large binary data from the DBMS may cause significant performance overheads and reduce user experience (Smith and Williams, 2000). As a result, the developers cache the images into ORM cache (Figure 3.1). Since the images are read-only, adding a *read-only* cache significantly improves application performance. Note that although this problem may also exist in other applications, the problem may have higher prevalence in ORM-based applications. Since ORM does not know whether image data is needed in the code, ORM will always fetch the image data from the DBMS under default configuration. This problem may be easily observed if developers manually write SQL queries and decide which columns should be retrieved from the database table.

Finally, we see that developers refactor how the database entity classes are designed and called to enhance application security. They do this by providing more validation rules (i.e., access control) to the user requests and to database entity object that are being retrieved from the DBMS.

We find that although there are common reasons for ORM and non-ORM code changes, some problems are more likely to cause of ORM code changes. Future studies may propose different techniques to detect such problems in order to assist developers with maintaining ORM-based applications.

Based on our manually studied samples of code changes, we find that compatibility, performance, and security problems are common reasons for ORM code changes. Thus, developing tools to detect such problems in ORM code could assist developers who are maintaining ORM-based applications.

4.6 Highlights and Implications of our findings

Our study has helped our industry partner recognize some key challenges associated with maintaining ORM code. Our study on the open source applications confirms that our findings are not specific to EA, and maintaining ORM code may be a wide-ranging concern. Although ORM frameworks are widely used in industry, many ORM-specific problems do not have a solution from the research world.

The highlights and implications of our findings are:

- ***ORM cannot completely encapsulate database access concerns***. We find that even though ORM tries to abstract database accesses, such abstraction cannot be completely encapsulated in objects. Future studies on the reasons that ORM cannot encapsulate database accesses in objects would help improve the design of ORM frameworks.

- ***ORM cannot completely abstract the underlying database technology***. Even though ORM ideally should ensure that the code is database-independent (i.e., porting an application to a different database technology should require no code changes), we still find several counterexamples. Future studies should examine the reasons that ORM cannot completely abstract the underlying database technology, and help developers create tools to migrate seamlessly to different database technologies.

- ***Although ORM code is frequently modified, there is a lack of tools to help prevent potential problems after ORM code changes***. In RQ2, we found that ORM code is modified more frequently than regular code, and some types of ORM code are modified even more frequently. However, since changes

to ORM model or query code may introduce runtime exceptions that affect the quality of an application, ORM code would benefit from type checking at compile time. Future research on providing automated tools to detect such problems can greatly reduce ORM maintenance efforts, and improve the quality of applications that make use of ORM frameworks.

- ***Traditional static code analyzers need to be extended to better capture the peculiarities of ORM code in order to find ORM-related problems***. ORM-related problems may be different, either syntactically or semantically, from the problems one may see in regular code (due to ORM's database abstraction). Thus, traditional static code analyzers, which usually do not consider the domain knowledge of ORM code, may not be able to detect these ORM-related problems without proper extensions. For example, FindBugs[1] is able to detect security problems in JDBC code, but FindBugs cannot detect such problems in ORM code without a proper extension. A recent study (Chen *et al.*, 2016b) shows that there are many database access code related problems that can be detected using static code analysis; however, existing tools and the research community have not put enough effort into detecting such domain-specific problems (i.e., related to database access). In general, due to the large number of available frameworks, a better option would be for framework developers to provide static code analyzers for the usage of their frameworks. Thus, developers who are using these frameworks can benefit from these code analyzers when developing their own applications.

[1] http://findbugs.sourceforge.net/

- ***Developers may benefit from tools that can automatically help them tune ORM performance configuration code***. We found that developers are more likely to change ORM code for performance reasons compared to regular code. However, in RQ2 we find that developers do not often change ORM configuration code. Prior studies (Dageville *et al.*, 2004; Yagoub *et al.*, 2008) from the database community show that tuning the performance of database-related code (e.g., SQL) is a continuous process, and needs to be done as applications evolve. Therefore, there may be many potential places in the code that require performance tuning. Hence, tools that can automatically change/tune ORM configuration code can be beneficial to developers. In Chapter 8, we follow up on this and propose an automated performance configuration tuning approach. We observe that finding an optimal configuration can significantly improve application performance.

Our industry partner is now aware of the high scatteredness of ORM code changes, and such changes are considered much riskier and require careful attention and review. In addition, our industry partner also recognizes the benefit of having tools that can automatically help refactoring and finding problems in ORM code.

4.7 Threats to Validity

We now discuss the threats to validity of our study.

Internal Validity. In this chapter, we study the characteristics and maintenance of ORM code. We discover that ORM code exhibits different patterns compared to

non-ORM code. However, we do not claim a casual relationship. There may be other confounding factors that influence our results (e.g., developers intentionally allocate more resources to the maintenance of files with ORM code, or the maintenance difficulties are caused by badly written code and not the design of ORM framework). Controlled user studies are needed to examine these confounding factors.

External Validity. To extend the generalizability of our results, we conduct our study on three open source applications and one large-scale industrial application. We choose these studied open source applications because they either have longer development history or are similar to the industrial application. There may be other similar Java applications that are not included in our study. Hence, future studies should examine additional applications to verify the generalizability of our findings. However, our current findings are already having an impact on how our industrial partner maintains ORM code.

We focus our study on JPA (Java ORM standard) because it is widely used in industry and is used by our industrial partner. However, our findings may be different for other ORM technologies (e.g, ActiveRecord or Django). Nevertheless, although the implementation details are different, these ORM frameworks usually share very similar concepts and configuration settings.

Construct Validity. We automatically scan the studied applications and identify ORM code. Therefore, how we identify ORM code may affect the result of our study. Although we use our expert knowledge and references to identify ORM code (ObjectDB, 2016b), our approach may not be perfect. We annotate a commit as an ORM-related commit if the commit modifies ORM code. Given the large

number of commits (over 10K), we have chosen to use an automated approach for commit classification. It is possible that some modifications in the commit are not related to ORM, or some commits may be related to refactoring activities. However, during our manual study in RQ3, we find that our approach can successfully identify ORM-related commits, and we believe our automated approach has a relatively high accuracy.

We compare ORM code with non-ORM code, but there may be many kinds of non-ORM code (e.g., GUI or network). However, since we are not experts in the studied applications, and there can be hundreds of different sub-components (depending on how we categorize non-ORM code), we choose to categorize code as ORM and non-ORM code. Ideally we would want to compare ORM code with other types of database access code (e.g., JDBC). However, most applications are implemented using only one database access technology, and it is not realistic to compare two different applications that use two database access technologies.

4.8 Chapter Summary

Object-Relational Mapping (ORM) provides a conceptual abstraction between DBMS and source code. Using ORM, developers do not need to worry about how objects in Object-Oriented languages should be translated to DBMS records. However, when cooperating with one of our industrial partners, we observed several difficulties in maintaining ORM code.

To verify our observations, we conducted studies on three open source Java applications, and we found that the challenges of maintaining ORM code is a wide ranging concern. Thus, understanding how ORM code is maintained is important,

and may help developers reduce the maintenance costs of ORM code. We found that 1) ORM code changes are more scattered and complex in nature, which implies that ORM cannot completely encapsulate database accesses in objects; future studies should study the root causes to help better design ORM code, especially in Java applications; 2) even though ORM ideally should ensure that the code is database-independent, we find that it is not always true; 3) ORM query code is often changed, which may increase potential maintenance problems due to lack of return type checking at compilation time; 4) traditional static code analyzers need to be extended to better capture the peculiarities of ORM code in order to find ORM-related problems; and 5) tools for automated ORM performance configuration tuning can be beneficial to developers. In short, our findings highlight the need for more in-depth research in the software maintenance communities about ORM frameworks (especially given the growth in ORM usage in software applications). Based on our empirical findings, in the result of the thesis, we propose approaches to detect performance anti-patterns in ORM code to help developers with ORM code maintenance (Chapter 5–7). In Chapter 8, we propose an approach to help developers automatically find an optimal ORM cache configuration.

CHAPTER 5

Statically Detecting ORM Performance Anti-patterns

As we find in Chapter 2 and Chapter 4, there is a limited tooling support for detecting performance problems in the database access code, and yet such problems are more likely to impact application performance. Thus, in this chapter, we propose an automated framework to detect ORM performance anti-patterns in the source code. Furthermore, as there could be hundreds or even thousands of instances of anti-patterns, our framework provides support to prioritize performance bug fixes based on a statistically rigorous performance assessment. We have successfully evaluated our framework on one open source and one large-scale industrial applications. Our case studies show that our framework can detect new and known real-world performance bugs and that fixing the detected performance anti-patterns can improve the application response time by up to 69%.

An earlier version of this chapter is published at the 36th International Conference on Software Engineering (ICSE), 2014. Hyderabad, India. Pages 1001–1012. (Chen *et al.*, 2014a)

5.1 Introduction

Object-Relational Mapping (ORM) provides developers a conceptual abstraction for mapping database records to objects in object-oriented languages. Through such mapped objects, developers can access database records without worrying about the database access and query details. For example, developers can call `user.setName("Peter")` to update a user's name in a database table. As a result, ORM gives developers a clean and conceptual abstraction for mapping the application code to the database management system (DBMS). Such abstraction significantly reduces the amount of code that developers need to write (Barry and Stanienda, 1998; Leavitt, 2000).

Despite ORM's advantages, using ORM frameworks may introduce potential performance problems. Developers may not be aware which source code snippets would result in a database access nor whether such access is inefficient. Thus, developers would not proactively optimize the database access performance, as we find in Chapter 4. For example, code that is efficient in memory (e.g., loops) may cause database performance problems when using ORM due to data retrieval overheads. In addition, developers usually only test their code on a small scale (e.g., unit tests), while performance problems would often surface at larger scales and may result in transaction timeouts or even hangs. Therefore, detecting and understanding the impact of such potential performance overhead is important for developers (Shang *et al.*, 2013). Nevertheless, as we find in Chapter 2, state-of-the-art static anti-pattern detection tools do not focus on detecting performance problems in database access code.

In this chapter, we propose an automated framework, which detects one type of

ORM performance anti-pattern, which we call one-by-one processing. Our framework can detect hundreds of instances of one-by-one processing anti-pattern based on static code analysis. Furthermore, to cope with the sheer number of the detected instances, our framework provides suggestions to prioritize bug fixes based on a statistically rigorous performance assessment (improvement in response time if the detected anti-patterns are addressed).

5.2 The Main Contributions of this Chapter

1. This is the first work that proposes a systematic and automated framework to detect and assess performance anti-patterns for applications developed using ORM.

2. Our framework provides a practical and statistically rigorous approach to prioritize the detected performance anti-patterns based on the expected performance gains (i.e., improvement in response time). The prioritization of detected anti-patterns is novel relative to prior anti-pattern detection efforts (Chis, 2008; Nistor *et al.*, 2013; Parsons and Murphy, 2004; Xiao *et al.*, 2013; Xu *et al.*, 2010b).

3. Through case studies on one open source applications (BroafLeaf commence (Commerce, 2013)) and one large-scale enterprise application (EA), we show that our framework can find existing and new performance bugs. Our framework, which has received positive feedback from the EA developers, is currently being integrated into the software development process to regularly scan the EA code base.

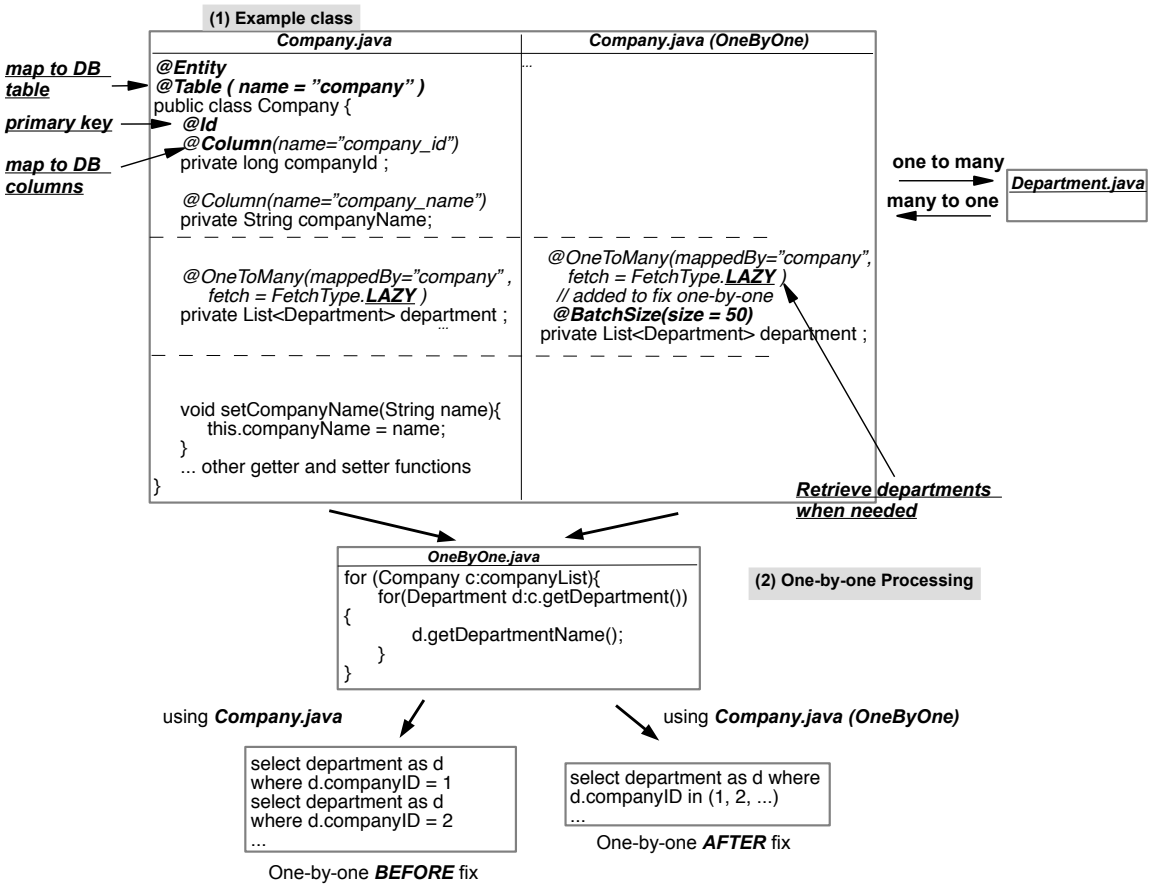


Figure 5.1: A motivating example. (1) shows the original class files and the ORM configurations; and (2) shows the modified `Company.java`, the one-by-one processing application code, and the resulting SQLs.

5.3 Motivating Examples

In this section, we present realistic examples to show how such ORM performance anti-patterns may affect the performance of an application. We develop a simple Java program as an illustration (Figure 5.1 (1)). The program manages a relationship between two classes (Company and Department), and provides a set of getter and setter methods to access and change the corresponding DB columns (e.g, setCompanyName).

In this example, there is a one-to-many relationship between Company and Department, i.e., one company can have multiple departments. This relationship is represented using annotation **@OneToMany** on the instance variable department in `Company.java` and **@ManyToOne** on the instance variable company in `Department.java` (details not shown in Figure 5.1 but very similar to `Company.java`). **@Column** shows which database column the instance variable is mapped to (e.g., companyId maps to column *company_id*), and **@Entity** shows that the class is a database entity class which maps to the database table specified in **@Table** (e.g., Company class maps to *company* table).

In the following subsections, we discuss how the performance anti-patterns may affect application performance, and show the performance difference before and after fixing the anti-patterns. We focus on one performance anti-pattern that is commonly seen in real-world code (Dubois, 2013; Wegrzynowicz, 2013) and that can possibly cause serious performance problems: *One-by-One Processing*, which repeatedly performs similar database operations in loops (Smith and Williams, 2001).

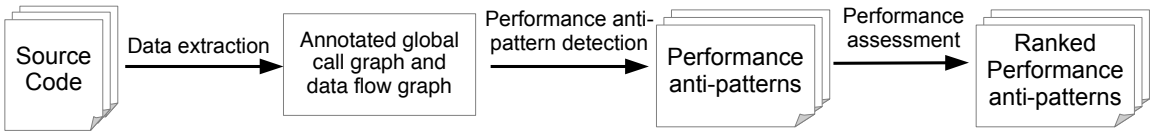


Figure 5.2: Overview of our static ORM performance anti-pattern detection and prioritization framework.

5.3.1 One-by-one Processing

Figure 5.1 (2) shows an example of the one-by-one processing anti-pattern. The one-by-one processing anti-pattern is a special case of the *Empty Semi Trucks* anti-pattern (Smith and Williams, 2001), which occurs when a large number of requests is needed to perform a task. In this chapter, we study the effect of such anti-pattern in the ORM context. `OneByOne.java` shows an application program that iterates through all the companies (companyList), and finds the names of all the departments in each company. If we execute `OneByOne.java` using the same ORM configurations in `Company.java`, it would generate one *select department* statement for *each* company object.

One way to solve this particular issue is to change the ORM configuration for retrieving department objects to the configuration in `Company.java (OneByOne)`. After adding a batch size (e.g., **@BatchSize**(size=50)) to the instance variable department, ORM will select 50 department objects in each batch. The fix reduces the number of SQL statements significantly, and could help improve the database performance. Note that the fix for one-by-one processing may vary in different situations (e.g., doing database updates in loops) and can be sometimes difficult.

To demonstrate the performance impact of one-by-one processing, we run the program shown in Figure 5.1 (2) with following setting: we populate the DBMS

with 300 records in the Company table and 10 records of Department for each record in the Company table. The response time before fixing the anti-pattern is 1.68 seconds, and the response time after the fix is 1.39 seconds (a 17% improvement).

Our motivating example shows that there is a significant performance improvement even in simple sequential database read operations after fixing instances of the performance anti-pattern.

5.4 Our Framework

This section describes our framework for detecting and prioritizing ORM performance anti-patterns. As shown in Figure 5.2, our framework consists of three phases: Data Extraction Phase, Performance Anti-pattern Detection phase and Performance Assessment phase. We first extract all the code paths, which involve database accesses. Then, we detect the performance anti-patterns among these database access code paths. Finally, we perform a statistically rigorous performance assessment so that developers can prioritize the performance bug fixes among hundreds of anti-patterns. In the rest of this section, we explain these three phases in detail. We use the motivating example from Section 5.3 to illustrate our approach.

5.4.1 Data Extraction

In this phase, we identify all the code paths, which involve database accesses. For each source code file, we extract the local call graphs, database access methods, and ORM configurations. Then, we combine the information from all the files and identify code paths (i.e., scenarios), which involve database accesses.

Extracting Database Access Methods, Local Call Graphs, and ORM Configurations

We first extract the call graphs and data flows in each file, then we identify all ORM-related information such as annotations and configurations. As shown in Chapter 3 and Section 5.3, ORM uses annotations to annotate a Java class as a database entity class or to setup different data retrieving strategies and configurations. We store the information about the ORM configuration of each database entity class and database instance variable, and identify methods that could result in database accesses (e.g., an ORM query or methods that retrieve an object from DBMS using the primary key). Using Figure 5.1 as an example, we store the relationship between Company and Department as one-to-many. We also mark the getter and setter methods (e.g., setDepartmentName), which access or modify database instance variables as database access methods.

Identifying Database Access Code Paths

We identify the code paths which involve database accesses. We accomplish this by:
1) Constructing Global Call and Data Flow Graphs: We build a global call graph for all the methods and we keep track of each object's data usage during its lifetime. Since methods in object-oriented languages are usually invoked through polymorphism, we connect the method call graphs using the corresponding methods in the subclasses for abstract classes or the implemented class for interfaces.
2) Marking Database Access Code Paths and Data Flows Using Taint Analysis: We use taint analysis, commonly used in computer security (Gollmann, 2011), to identify code paths and data flows, which involve database accesses. Taint analysis

keeps track of a possibly malicious variable (e.g., variable V), and if V is used in an expression to manipulate another variable M, M is also considered suspicious. Similarly, if a method in a call path contains a database access method that is determined by our first step, we mark all the methods in the code path as database access methods. For example, in a call path, method A calls method B and method B calls method C. If method C is a database access method, we consider the code path of A $\rightarrow$ B $\rightarrow$ C as a database access code path. We perform taint analysis statically by computing the node reachability across methods in the global method call graph.

5.4.2 Performance Anti-pattern Detection

With identified database access code paths and data flows, we use a rule-based approach to detect ORM performance anti-patterns. Different anti-patterns are encoded using different rules. In this chapter, we focus on detecting one of the most pervasive ORM anti-patterns, but new anti-patterns can be integrated to the framework by adding new detection rules (as illustrated in Chapter 6). Section 5.6 has more discussions about extending our framework.

Detecting One-by-one Processing Anti-patterns: To detect one-by-one processing anti-patterns, the framework first identifies the methods that are directly called within both single and nested loops (we consider all kinds of loops, such as *for*, *while*, *foreach*, and *do while*). If a directly-called method is a database access method, the framework simply reports it as an one-by-one processing anti-pattern. The framework also traverses all the child nodes in the method call graphs of the methods that are called in loops. We analyze the child nodes across multiple methods, and determine whether there is a node in the call graph that may access the

DBMS. If so, the framework reports the call path that contains the node as an instance of one-by-one processing anti-pattern.

Due to ORM configurations, some database access methods may not always access the database. In such cases, we do not report them as anti-patterns. For example, if Department is eagerly retrieved by Company, retrieving Company will automatically retrieve Department. Therefore, accessing Department through Company will not result in additional database accesses, and will not be reported by our framework. If a database access method in a loop is already being optimized using some ORM configurations (i.e., fetch plan is either ***Batch***, ***SubSelect***, or ***Join***) (Community, 2016), we do not identify it as a performance anti-pattern.

5.4.3 Performance Assessment

Previous performance anti-pattern detection studies generally treat all the detected anti-patterns the same and do not provide methodologies for prioritizing the anti-patterns (Chis, 2008; Nistor *et al.*, 2013; Parsons and Murphy, 2004; Xiao *et al.*, 2013; Xu *et al.*, 2010b). However, as shown in the case studies (Section 5.5), there could be hundreds or thousands of performance anti-pattern instances. Hence, it is not feasible for developers to address them all in a timely manner. Moreover, some instances of anti-patterns may not be worth fixing due to the high cost of the fix and small performance improvement. For example, if a one-by-one processing anti-pattern always processes a table with just one row of data, there is little improvement in fixing this particular anti-pattern instance.

In this phase, we assess the performance impact of the detected anti-patterns through statistically rigorous performance evaluations. The assessment result may

not be the actual performance improvement after fixing the anti-patterns, but may be used to prioritize the fixing efforts. We repeatedly measure and compare the application performance before and after fixing the anti-patterns by exercising the readily available test cases. Anti-patterns, which are expected to have big performance improvements (e.g., 200% performance improvement) after the fixes, will result in higher priorities. The rest of this subsection explains the three internal parts in performance assessment phase: (1) exercising performance anti-patterns and calculating test coverage, (2) assessing the performance improvement after fixing the anti-patterns, and (3) statistically rigorous performance evaluations.

Part 1 - Exercising performance anti-patterns and calculating test coverage

One way to measure the impact of performance anti-patterns is to evaluate each anti-pattern individually. However, since application performance is highly associated with run-time context and input workloads (Goldsmith *et al.*, 2007; Xiao *et al.*, 2013; Zaparanuks and Hauswirth, 2012), we need to assess the impact of performance anti-patterns using realistic scenarios and workloads.

Since performance anti-patterns are detected in various software components, it is difficult to generate workloads to exercise all the performance anti-patterns manually. Therefore, we use the readily available performance test cases to exercise the performance anti-patterns. For the applications that do not have performance test cases, we use integration test cases as an alternate choice. Although integration test cases may not be designed for testing performance critical parts of an application, they are designed to test various *features* in an application (i.e., use case tests), which may give a better test coverage. In short, performance and integration test cases group source code files that have been unit tested into larger units as suites

(e.g., features or business logic), which better simulate application workflows and user behaviours (Binder, 2000).

Since we do not have control over which instances of performance anti-patterns are exercised by executing the test cases, it is important to know how many performance anti-patterns are covered by the tests. We use a profiler to profile the code execution paths of all the tests, and use the execution path to calculate how many instances of performance anti-patterns are covered in the tests.

Part 2 - Assessing performance improvement after fixing anti-patterns

In this part, we describe our methodology to assess the performance improvement of fixing the one-by-one processing anti-pattern.

Fixing one-by-one processing can be much more complicated than what we have shown in Section 5.3. One common fix to one-by-one processing is using batches. However, for example, performing a batch insert to the *Department* table in ORM may require writing specific ORM SQL queries, such as "*insert into Department (name) values ('department1'), ('department2')* ... ", to replace ordinary ORM code. As a result, manually fixing the anti-patterns of one-by-one processing requires a deep knowledge about the structure, design and APIs of an application. Due to the complexity, it is very difficult to fix all the detected one-by-one processing anti-patterns automatically. Therefore, we follow a similar methodology by Jovic *et al.* (2011) to assess the anticipated application performance after fixing the one-by-one processing anti-patterns.

As shown in Section 5.3, the slow performance in the one-by-one processing anti-patterns is mainly attributed to the inefficiency in the generated SQL statements. The one-by-one processing anti-patterns in ORM generate repetitive SQL

statements with minor differences. The repetitive SQL statements can be optimized using batches, which reduce a large amount of query preparation and transmission overheads. Therefore, the performance measure for executing the optimized SQL statements could be a good estimate for the anticipated application performance after fixing the one-by-one processing anti-patterns. To obtain the generated SQL statements, we use a SQL logging library called *log4jdbc* (Bloom, 2013) to log the SQL statements and SQL parameter values. *log4jdbc* simply acts as an intermediate layer between JDBC to the DBMS, which relays the SQL statements to the DBMS and outputs the SQL statements to log files.

We detect the repetitive SQL statements generated by ORM, and execute the SQL statements in batches using Java Database Connectivity (JDBC), which assess the performance impact after fixing the performance anti-pattern. Note that since JDBC does not support batch operations for *select*, we exclude *select* in our assessment. We execute the original non-optimized and optimized SQL statements separately and compare the performance differences in terms of response time.

Part 3 - Statistically rigorous performance evaluation

Performance measurements suffer from instability, and may lead to incorrect results if not handled correctly (Arnold *et al.*, 2002; Georges *et al.*, 2007; Kalibera and Jones, 2013). Thus, a rigorous study is required when doing performance measurements (Kalibera and Jones, 2013). Therefore, our framework does repeated measurement and computes effect sizes of the performance improvement (i.e., quantifies the increase in response time statistically) to overcome the instability problem.

Repeated measurements: Georges *et al.* (2007) recommend computing a confidence interval for repeated performance measurements when doing performance

evaluation, since performance measurements without providing measures of variation may be misleading and incorrect (Kalibera and Jones, 2013). We repeat the performance tests 30 times (as suggested by Georges *et al.* (2007)), and record the response time before and after doing fixes. We use *Student's t-test* to examine if the improvement is statistically significant different (i.e., p-value i 0.05). A p-value i 0.05 means that the difference between two distributions is likely not by chance. Then, we compute the mean response time and report the 95% confidence interval. A *t-test* assumes that the population distribution is normally distributed. According to the central limit theorem, our performance measures will be approximately normally distributed if the sample size is large enough, since the noise in each of our test run is independent (we reset the application state after each test run) (Georges *et al.*, 2007; Moore *et al.*, 2009).

Effect size for measuring the performance impact: We conduct a rigorous performance improvement experiment by using *effect sizes* (Kampenes *et al.*, 2007; Nakagawa and Cuthill, 2007). Unlike *t-test*, which only tells us if the differences of the mean between two populations are statistically significant, effect sizes quantify the difference between two populations. Researchers have shown that reporting only the statistical significance may lead to erroneous results (Kampenes *et al.*, 2007) (i.e., if the sample size is very large, p-value can be small even if the difference is trivial). We use *Cohen's d* to quantify the effects (Kampenes *et al.*, 2007). *Cohen's d* measures the effect size statistically, and has been used in prior engineering studies (Kampenes *et al.*, 2007; Kitchenham *et al.*, 2002). *Cohen's d* is defined as:

$$\textit{Cohen's } d = \frac{\bar{x}_1 - \bar{x}_2}{s}, \tag{5.1}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the mean of two populations, and s is the pooled standard

deviation (Hartung *et al.*, 2011).

The strength of the effects and the corresponding range of *Cohen's d* values are (Kampenes *et al.*, 2007):

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } \textit{Cohen's } d \leq 0.2 \\ \text{small} & \text{if } 0.2 < \textit{Cohen's } d \leq 0.5 \\ \text{medium} & \text{if } 0.5 < \textit{Cohen's } d \leq 0.8 \\ \text{large} & \text{if } 0.8 < \textit{Cohen's } d \end{cases}$$

We output the performance anti-patterns, ranked by their effect sizes, in an HTML report. The report contains the database assess methods and the code path of the detected one-by-one processing anti-patterns.

5.5 Case Study

In this section, we apply our framework on one open source application (BroafLeaf commence) and one large-scale closed-source enterprise application (EA). We seek to answer the following two research questions:

RQ1: What is the performance impact of the detected anti-patterns?

RQ2: How do performance anti-patterns affect application performance at different input scales?

Each research question is organized into three sections: Motivation, Approach and the Results. Table 5.1 shows the statistics of the studied applications.

All of our studied applications use JPA for ORM and follow the "Model-View-Controller" design pattern (Krasner and Pope, 1988). Broadleaf is a large open

Table 5.1: Statistics of the studied applications and number of detected anti-pattern instances.

	Total lines of code (K)	No. of files	No. of 1-by-1 processing
Broadleaf 3.0	206K	1,795	228
EA	>300K	>3,000	>10

source e-commerce application that is used in many commercial companies worldwide for building online transaction platforms. EA supports a large number of users concurrently and is used by millions of users worldwide on a daily basis. We sought to use open source applications, in addition to the commercial application, so others can verify our findings and replicate our experiments on the open source applications, as we are not able to provide access to the EA.

RQ1: What is the performance impact of the detected anti-patterns?

Motivation. Application performance is highly associated with run-time context and input workloads (Goldsmith *et al.*, 2007; Xiao *et al.*, 2013; Zaparanuks and Hauswirth, 2012). Rarely or never executed anti-patterns would have less performance impact compared to frequently-executed ones. Therefore, we use test cases to assess the performance impact instead of executing the anti-patterns individually. In this research question, we want to detect and prioritize performance anti-patterns by exercising different features of an application using our proposed framework (Section 5.4).

Approach. test case Since performance problems are usually revealed under large data sizes (Goldsmith *et al.*, 2007; Jin *et al.*, 2012), we manually increase the data sizes in these test cases and write a data loader for loading data into the DBMS. Our

Table 5.2: Performance assessment result for one-by-one processing. Tests with p-value < 0.05 have statistically significant performance improvement (marked in bold). Numbers in the parentheses are the percentage reduction in response time.

Application	Test Case Description	No. One-by-one Processing Covered	μ Before (sec)	μ After (sec)	Statistical Significance (p-value)	Effect size
Broadleaf	Customer Phone	61	1.00±0.30	1.09±0.33(-0.09%)	0.68	0.10 (trivial)
	Offer Service	67	1.23±0.47	1.08±0.34(-12%)	0.60	0.13 (trivial)
	Shopping Cart	**52**	**21.46±1.64**	**14.58±0.42(-32%)**	**<<0.001**	**2.07(large)**
	Checkout	**109**	**13.25±0.30**	**10.63±0.66(-20%)**	**<<0.001**	**1.86 (large)**
	Customer Addr.	**61**	**1.49±0.33**	**1.08±0.10(-27%)**	**<0.05**	**0.59 (medium)**
	Customer	61	1.11±0.42	0.95±0.32(-15%)	0.54	0.15 (trivial)
	Order	**104**	**13.12±0.27**	**10.20±0.18(-22%)**	**<<0.001**	**4.54 (large)**
	Offer	62	1.23±0.56	0.95±0.25(-22%)	0.37	0.23 (small)
	Payment Info	61	1.08±0.31	1.16±0.38(+8%)	0.74	0.08 (trivial)
EA	**Multiple Features**	> 10	—	**improved by 69%**	**<<0.001**	**55.3 (large)**

framework repeatedly exercises the test cases and computes the effect sizes of the performance impact. Moreover, we measure the performance impact before and after fixing all the anti-patterns in each test case, separately. We use the percentage reduction in response time, statistical significance of such reduction, and effect size to measure performance impact (i.e., whether there is an actual difference and how large the effect is).

Results. *Anti-pattern detection results:* Table 5.1 shows the anti-pattern detection result. In Broadleaf, our framework detected a total of 308 instances of one-by-one processing anti-pattern. Since a large number of anti-pattern instances is detected, we only emailed the top 10 instances of high impact performance anti-patterns to the developers. We are currently waiting for their reply. Due to non-disclosure agreement (NDA), we cannot present the exact numbers of detected anti-patterns in EA. However, we can confirm that our framework is able to detect many of the existing and new performance problems in EA.

Performance benefits of removing one-by-one processing. Table 5.2 shows the performance impact of one-by-one processing in each test case, and the assessed performance improvement.

In Broadleaf, one-by-one processing anti-patterns have a statistically significant performance impact in 4 out of the 9 test cases, and the effect sizes are at least medium (0.59–4.54). The assessed response time reduction is from 20–32%.

One-by-one processing anti-patterns have a non-statistically significant impact in other test cases. These test cases have one common behaviour: very short response time. The results indicate that when the response time of a program is small, adding batches will not give much improvement. This also shows that not all anti-patterns are worth fixing.

Although we cannot show the mean response time and confidence interval in EA, the assessed response time reduction is high (69%) and the effect size is large (55.3). By using batch operations, the performance of EA was improved significantly.

Our performance assessment results show that performance anti-patterns have a statistically significant performance impact in 5/10 test cases with effect sizes varying from medium to large. We find that fixing the performance anti-patterns may improve the response time by up to 69%.

RQ2: How do performance anti-patterns affect application performance at different input scales?

Motivation. In RQ1, we manually change the data sizes to large to study the impact of performance anti-patterns. However, populating large volumes of data into

Table 5.3: Performance assessment result for different scales of data sizes. We do not show the effect size for the tests where the performance improvements are not statistically significant (i.e., p-value >= 0.05).

Application	Test Case Description	Effect Sizes for Different Input Sizes		
		small	medium	large
Broadleaf	Shopping Cart	0.88	1.77	2.08
	Checkout	–	0.55	1.86
	Customer Addr.	–	0.51	0.59
	Order	–	1.46	4.54
EA	Multiple Features	16.2	21.8	55.3

the DBMS requires a long time, and a database expert is needed to ensure that the generated data satisfies all the database schema requirements (Kapfhammer *et al.*, 2013). In this research question, we study whether we still have the same prioritization ranking of the performance anti-patterns using smaller data sizes.

Approach. We focus only on the test cases which yield anti-patterns with statistical significant performance impact in RQ1. We manually modify the test cases to change the data sizes to medium and small as opposed to big data sizes in RQ1. We reduce the data sizes by a factor of 2 at each scale (e.g., medium data size is 50% of large data size and small data size is 50% of the medium data size). Finally, we re-run the performance tests to study how the performance impact and effect size change at smaller scales.

Results. Table 5.3 shows the performance impact assessment of one-by-one processing at different scales. In general, one-by-one processing anti-patterns in Broadleaf have a higher performance impact (i.e., larger effect sizes) when the data sizes increase, because the data sizes directly affect the number of iterations in loops. However in most test cases, these one-by-one processing anti-patterns still have a

statistically significant impact at smaller scales. We find that only three test cases do not have a significant performance impact when the data size is small, but all test cases have a significant performance impact when the data size is medium (effect size 0.51–1.77). We find a similar trend in EA, where the effect size increases as data size increases. We can still identify performance problems in these test cases using small to medium data sizes.

We find that the priority of the performance anti-patterns at different scales is consistent, i.e., the rank of the effect sizes in different test cases is consistent across different input data scales. For example, the rank of the effect sizes for test cases using medium and large data sizes is the same except for the ranks of the Shopping cart test and the Order test, which are swapped. As a result, if the generation of large dataset takes too long or takes too much effort, we are still very likely to reproduce the same set of severe performance problems using a smaller dataset.

We find that the prioritization of performance anti-patterns when exercised on medium scale data size is very similar to the large data size. This results show that developers may not need to deal with all the difficulties of populating large data into the DBMS to reveal these performance problems.

5.6 Discussion

In this section, we discuss the accuracy of our performance assessment, extensions of our framework, and initial developer feedback.

The accuracy of our one-by-one processing performance assessment methodology. In our performance assessment methodology for one-by-one processing, we

measure the response time of the original non-optimized and optimized SQL statements instead of directly fixing the code. To study the accuracy of this methodology, we develop a simple program with a known one-by-one processing anti-pattern (example in Section 5.3) for evaluation.

We populate a DBMS with $100,000$ department names in all companies and verify using the example in Figure 5.1 (2). We fix the one-by-one processing pattern in the code by writing SQL statements using JPA-specific query language. The original code takes about 24.67 ± 0.84 seconds. Fixing the code results in a mean response time of 5.36 ± 0.06 seconds, and the assessment shows a mean response time of 10.42 ± 0.09 seconds. The experiment shows that our assessment methodology may be an over-estimate but can achieve a comparable performance improvement (78.3% v.s. 57.8%) to assess the impact of the anti-patterns. In the future, we plan to investigate automated performance refactoring approaches for fixing the one-by-one processing anti-patterns.

Framework extension. In this chapter, we proposed a rule-based approach, which detects and prioritizes one of the most pervasive ORM performance anti-patterns. Similar to any other pattern detection work, our framework cannot detect unseen performance anti-patterns. However, our framework could easily be extended by encoding other performance anti-patterns. In Chapter 6, we discuss how an extension of the framework is adopted into practice.

Initial Developer Feedback. We received positive feedback from developers and performance testers in EA. The framework is able to help them narrow down the performance problems and find potential bottlenecks. The framework is now integrated into the daily development processes of EA.

5.7 Threats to Validity

In this section, we discuss the threats to validity.

5.7.1 External Validity

We have only evaluated our framework on three applications. Some of the findings might not be generalizable to other applications. Although the studied applications vary in sizes and domains, other similar applications may have completely different results. Future work should apply our framework to more applications and even different programming languages (e.g., C#).

5.7.2 Construct Validity

Detection approach. We use static analysis for detecting performance anti-patterns. However, static analyses generally suffer from the problem of false positives, and our framework is no exception. For example, a detected performance anti-pattern may be seldom or never executed due to reasons like unrealistic scenarios or small input sizes. Therefore, we provide a performance assessment methodology to verify the impact and prioritize the fixing of the detected performance anti-patterns.

Experimental setup. We exercise the performance anti-patterns using test cases, so we do not have control over which performance anti-patterns will be exercised. As a result, our performance impact assessment study only applies to the exercised performance anti-patterns. However, our performance impact assessment methodology is general, and can be used to discover the impact of performance anti-patterns in different software components and to prioritize QA effort.

We manually change the data sizes to study how the impact of performance anti-patterns changes in different scales. Changing the data loader in the code and loading data in the DBMS require a deep understanding of the application's structure and design of each test case. Although we studied the code in each test case and database schemas carefully, it is still likely that we did not change the inputs that are directly associated with the performance anti-patterns, or that the data does not generate representative workloads. However, case studies show that we can still detect a similar set of high impacting performance anti-patterns using different sizes of data.

Fixing performance anti-patterns and performance assessment. Fixing some performance anti-patterns may require API breaks and redesign of the application. Therefore, fixing them may not be an easy task. For example, to achieve maximal performance improvement, sometimes it is necessary to write SQL statements in ORM (Linwood and Minter, 2010). If the anti-pattern is generating many small database messages, which cause transmitting overheads and inefficient bandwidth usage, the solution is to apply batching (Smith and Williams, 2003). In addition, different implementations of ORM support different ways to optimize performance. As a result, we provide a performance assessment methodology for assessing the performance impact. We use a similar methodology as Jovic *et al.* (2011) to measure the performance impact of one-by-one processing. Although our performance assessment methodology may not give the exact performance improvement and there may be other ways to fix the performance anti-patterns, we can still use the

assessment result to prioritize the manual verification and performance optimization effort. We can further reduce the overheads of running the performance assessment approach using the result of our static analysis, and focus only on the parts of the application that are prone to performance anti-patterns.

It is possible that the performance fixes may have contradicting result in different use cases. For example, in some cases using a fetch type of *EAGER* may yield a better performance, but may yield performance degradation in other cases. However, since ORM provides a programming interface to change the configuration and fetch plans dynamically, the problem can be solved by developers easily.

5.8 Chapter Summary

Object-Relational Mapping (ORM) provides a conceptual abstraction between the application code and the DBMS. ORM significantly simplifies the software development process by automatically translating object accesses and manipulations to database queries. Developers can focus on business logic instead of worrying about non-trivial database access details. However, there are hidden costs when using these ORM frameworks, as we find in Chapter 4. Writing ORM code incorrectly or inefficiently might lead to performance anti-patterns, causing transactions timeout or hangs in large-scale software applications.

In Chapter 2, we find that existing static anti-pattern detection tools do not support detecting performance problems in database access code. Therefore, in this chapter, we propose a framework, which can detect and prioritize instances of ORM performance anti-patterns. We applied our framework on two software applications: one open-source and one large enterprise applications. Case studies show

that our framework can detect hundreds or thousands instances of performance anti-patterns, while also effectively prioritizing the fixing of these anti-pattern instances using a statistically rigorous approach. Our static analysis result can further be used to guide dynamic performance assessment of these performance anti-patterns, and reduce the overheads of profiling and analyzing the entire application. We find that fixing these instances of performance anti-patterns can improve the applications' response time by up to 69%. In the rest of the thesis, we first discuss an extension to our static anti-pattern framework and the experience we learned when adopting the framework in practice. Then, to address the limitation of static analysis, we propose another anti-pattern detection approach using dynamic analysis. Finally, to address the problem that we find in Chapter 4, where developers rarely tune ORM performance configurations, we propose an automated approach to help developers find an optimal ORM cache configuration.

CHAPTER 6

Adopting Anti-pattern Detection Framework in Practice

In Chapter 5, we propose a static performance anti-pattern detection framework, and our framework is well received by our industrial partners. In this chapter, we document our industrial experience over the past few years on finding anti-patterns of database access code, implementing an anti-pattern detection tool, and integrating the tool into daily practice. We discuss the challenges that we encountered and the lessons that we learned during integrating our tool into the development process. We also provide a detailed discussion of five framework-specific database access anti-patterns that we found. We hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language anti-patterns and associated detectors.

An earlier version of this chapter is published at the 38th International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP), 2016. Austin, Texas. Pages 71–80. (Chen *et al.*, 2016b)

6.1 Introduction

DUE to the emergence of cloud computing and big data applications, modern software applications become more dependent on the underlying database management systems (DBMSs) for providing data management and persistency. As a result, DBMSs have become the core component in such database-centric applications, with DBMSs usually interconnecting other components. Thus, due to the complexity of how developers interact with the DBMSs, it can be difficult to discover and detect problems (e.g., anti-patterns) in database access code.

Since the exact behaviour of these DBMSs are often blackboxes to developers, writing high quality test cases that can uncover all database access problems is nearly impossible. In additional to testing, developers usually use static anti-pattern detection tools, such as FindBugs (Hovemeyer and Pugh, 2004) and PMD (2016), to provide a complete coverage of the entire application. However, these anti-pattern detection tools usually only provide patterns for detecting general code bugs, but cannot detect domain-specific anti-patterns that are associated with accessing databases. For example, modern applications usually leverage different frameworks to abstract database access to speed up development time and reduce maintenance difficulty. Thus, using these frameworks incorrectly may introduce more domain-specific anti-patterns. In addition, existing static anti-pattern detection tools usually rely on scanning the binary files, but many database access anti-patterns that we see in practice require parsing specialized annotations in the code or analyzing external SQL scripts.

In Chapter 5, we implemented a prototype tool to detect a database access anti-pattern in collaboration with industry. Our research-based domain-specific static anti-pattern detection tool, *DBChecker*, received very positive feedback from developers, and was adopted and integrated as part of the day-to-day development processes of the industrial application. We have worked closely with developers in order to ease the adoption of our tool. However, during such process, we encountered challenges and learned lessons that are associated with how to successfully make practitioners adopt a research-based domain-specific static anti-pattern detection tool. In this chapter, we document and discuss the challenge and lessons learned. We believe that our experiences can help other researchers improve anti-pattern detection tools and ensure a smoother adoption process of their tools in practice.

This chapter also documents five additional framework-specific database access anti-patterns that we observed while working on several industrial applications. Our goal is to give readers concrete examples of framework-specific anti-patterns. Since most modern applications are leveraging frameworks, framework-specific anti-patterns can have a large impact in practice. For example, the anti-pattern that we study in Chapter 5 can have significant performance impact and is common in database-centric applications (Yan *et al.*, 2016). Hence, we hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language anti-patterns and associated detectors.

6.2 The Main Contributions of this Chapter

1. We provide an experience report that discusses the lessons that we learned on discovering, locating, and detecting the anti-patterns, and the challenges that we encountered when adopting our anti-pattern detection tool in practice.

2. We provide detailed documentation on the root causes and impact of five database access anti-patterns that we have seen in the studied industrial applications over the past few years.

6.3 Related Work

In this section, we discuss related work to our study.

6.3.1 Integrating Code Analysis Research Tools into Practice

In addition to this chapter, there are some prior studies discussing the challenges associated with adopting static anti-pattern detection research tool into practice. Johnson *et al.* (2013) interview 20 developers regarding the challenges that they see when using static anti-pattern detection tools. Johnson *et al.* (2013) find that tool configuration (e.g., filtering mechanism), integration with development workflow, and report formatting affect developers' willingness to use a static anti-pattern detection tool. Ayewah *et al.* (2007) evaluate the generated warnings by FindBugs on production software. They found that FindBugs finds many true bugs with little or no functional impact, and there is a need for prioritizing high impact defects.

Nanda *et al.* (2010) discuss how they use an online portal to help improve the adoption of static anti-pattern detection tools at IBM. The portal provides cloud-based code scanning and allows developers to communicate by adding discussion to the static anti-pattern detection reports. Smith *et al.* (2015) conduct a user study on the questions that developers ask when using static anti-pattern detection based security tools. They found that the security tools should provide better reporting systems to help developers detect the problems.

In this chapter, we focus on database access anti-patterns, which have different characteristics than general anti-patterns. For example, due to the differences in the nature of the accessed database tables (e.g., size), some detected instances of anti-patterns may be more severe in practice. Since adopting research results in practice can be very challenging (Lo *et al.*, 2015), we discuss the challenges and lessons that we learned when adopting our static anti-pattern detection tool in practice.

6.4 Background

In this section, we briefly discuss the industrial applications that we use, and the background story behind creating a tool to detect database access anti-patterns.

Studied Applications. Due to non-disclosure agreement (NDA), we cannot give the exact details of the applications. However, the industrial applications are very large in sizes (millions of lines of code), support a large number of users concurrently, and are used by millions of users worldwide on a daily basis. Below, we discuss the two main technologies that these industrial applications often use for accessing DBMS and managing database transactions.

There is a slew of similar technologies used in practice today by researchers and developers. Hence, the discussed patterns and our experiences are general, and are not specific to a particular technology; instead, the patterns are due primarily to the interaction between application code and database. We discuss the technology below because of their popularity and our need to provide concrete examples throughout the chapter, so the reader can better grasp the raised concerns and the documented patterns.

Transaction Management Using Spring. Spring (SpringSource, 2016) is a widely used framework for database transaction management, based on an aspect-oriented approach. A recent survey (ZeroturnAround, 2014) shows that Spring is the most commonly used Java web framework (more than 40% of developers use Spring). Spring abstracts database transaction management code using annotations. For example:

```
1 @Transactional
2 public void performBusinessTransaction(){
3         ...
4 }
```

In this simple code example, by adding the annotation **@Transactional**, the method performBusinessTransaction and all the timemethods called within it will be executed in a single database transaction. Thus, developers can avoid writing boilerplate code, instead they can focus on the business logic of the application.

In practice, we see some database access anti-patterns that are related to how a transaction is configured when using Spring. For example, a transaction can have the default configuration, where a transaction will be created if the annotated

method is not already in a transaction. If the annotated method is already within a transaction (e.g., one of the caller methods is also annotated with **@Transactional**), the method would be executed in the parent transaction and will not create a new transaction. If the annotation has the property **@Transactional(REQUIRES_NEW)**, then a new transaction will always be created, and the parent transaction will be suspended until the newly created transaction is completed. If the annotation has the property **@Transactional(NOT_SUPPORTED)**, then the parent transaction will be suspended until the annotated method returns. In practice, we have seen incorrect uses of such configuration that cause functional or non-functional problems (e.g., deadlocks, feature bugs, and scalability issues).

6.5 Challenges and Lessons Learned

We encountered many bugs that are associated with accessing a DBMS when developing large-scale industrial applications. In our previous Chapter 5, we derived some anti-patterns based on such bugs, and implemented a prototype static anti-pattern detection tool. Our tool received very positive feedback from developers, and attracted interests from various development teams. After active discussion and cooperation with the developers, we received many additional database access anti-patterns that the developers have seen over the years in the field. Based on our experience (Chapter 5), database access anti-patterns can have significant impact on the application quality. As these database access anti-patterns cannot be detected using readily available static anti-pattern detection tools such as FindBugs or PMD, we implemented the detection algorithms and integrated them into our *DBChecker* tool. However, although developers see value in our database access

anti-pattern detection tool, we encountered many roadblocks when adopting this line of research into practice. As a result, we feel that, in addition to discussing new database access anti-patterns that we have seen since our prior work, documenting the challenges that we encountered and the lessons that we learned can further help researchers create tools that have a higher chance to be adopted in practice, and can reduce the gap between static anti-pattern detection research and practice.

Below, we provide detailed discussions on the challenges that we encountered when adopting our tool to detect database access anti-patterns. For each challenge, we provide the description and impact of the challenge, our solutions to address the challenges, and the lessons that we learned.

C1: Handling the Large Size of Detection Results

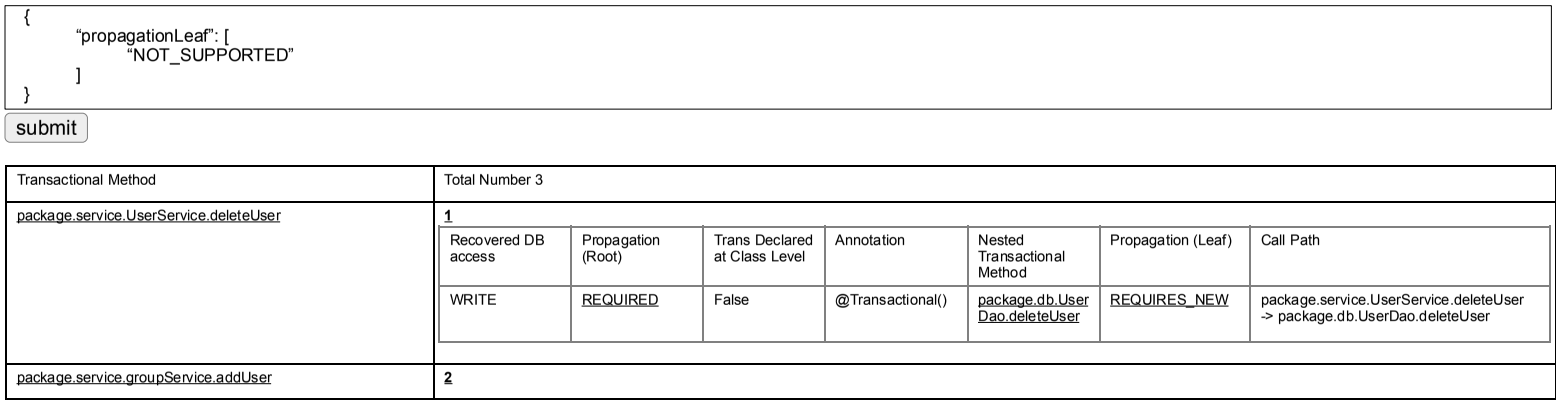
Challenge Description. A common challenge when using static anti-pattern detection tools is that these tools usually report a large number of anti-pattern instances that overwhelm the developers (Johnson *et al.*, 2013; Shen *et al.*, 2011), and our tool is not an exception. However, we found that not all of the detected anti-pattern instances are real problems, since some of them may be false positives. In addition, to the best of our knowledge, there is no prior study that discusses the integration of static performance anti-pattern detection tool in practice. We found that many detected performance anti-patterns are true bugs, but their impact may be too small to be relevant.

Challenge Impact. Showing all the detected anti-pattern instances to the developers at once would quickly reduce developers' interest and trust in the tool. In addition, developers usually only have limited time and resources to investigate a

portion of the detected anti-pattern instances. So it is important to highlight the anti-pattern instances that have the highest impact. Therefore, helping developers make fast decision on whether a detected anti-pattern instance is a false positive, and how to prioritize their efforts on reviewing the detection results is very important for effective QA resource utilization.

Solutions to Handling the Large Size of Detection Results and Lessons Learned. We found that in many cases, the detected anti-pattern instances may not have the same impact, even though the anti-pattern instances belong to the same pattern. For example, a detected anti-pattern instance that is related to a frequently accessed database table can have a larger impact in practice, compared to anti-pattern instances related to rarely accessed tables. We also found that we need to consider the size the table that the database access code is accessing, since large data sizes can increase the impact of an anti-pattern instance (Goldsmith *et al.*, 2007). However, it is impossible to get the above-mentioned information using static analysis, but the information can greatly help reduce developers' effort on inspecting the static analysis results. Based on the feedback we received from developers, we have integrated several functionalities into *DBChecker* that helped us improve tool adoption and acceptance.

Grouping Detected Anti-pattern Instances. Grouping the detected anti-pattern instances allows developers to allocate more resources to important components of the application. Figure 6.1 shows an example detection report of the *nested transaction* anti-pattern. We group the detected anti-pattern instances according to the source (i.e., packages or root causes) to which they belong, such that developers can focus on features that are more important (similar features are usually located in



```
{
        "propagationLeaf": [
                "NOT_SUPPORTED"
        ]
}
```

submit

Transactional Method	Total Number 3
package.service.UserService.deleteUser	**1** Recovered DB access: WRITE Propagation (Root): REQUIRED Trans Declared at Class Level: False Annotation: @Transactional() Nested Transactional Method: package.db.UserDao.deleteUser Propagation (Leaf): REQUIRES_NEW Call Path: package.service.UserService.deleteUser -> package.db.UserDao.deleteUser
package.service.groupService.addUser	**2**

Figure 6.1: An example detection report for the *nested transaction* anti-pattern.

the same package and affected by the same problem). In order to identify whether the detected anti-pattern instance is real and has a sizeable impact in a timely manner, our tool also recommends the experts that should investigate the anti-pattern instance, based on the developer who last modified the method, in which the detected anti-pattern instance was found.

To help developers allocate quality assurance efforts, we group the detected anti-pattern instances according to their locations in the code.

Prioritizing Detected Anti-pattern Instances. We found that some of the studied database access anti-patterns may have varying severity in different use cases, especially for performance related anti-patterns. For example, if an anti-pattern is related to relationships between two database tables, a many-to-many relationship would be more severe than one-to-one (Chen *et al.*, 2014a, 2016d). In the case of eagerly fetching data from the DBMS, a one-to-many relationship between two tables means that if we retrieve data of one user from the DBMS, we will also eagerly retrieve data of all the groups to which the user belongs. Thus, the anti-pattern instance becomes more severe compared to the same pattern but with a one-to-one relationship. Hence, it is important to prioritize the detected anti-pattern instances according to their potential severity to reduce the inspection effort of developers. We also find that providing a sorting mechanism in the detection report can further help developers allocate QA resources. Since some database tables are accessed more frequently or have more data, detected anti-pattern instances that involve those tables should be ranked higher. Developers should be able to choose the database table of interest, and the report would prioritize the detected anti-pattern instances that are related to those tables.

To help developers allocate quality assurance efforts, we prioritize the detected anti-pattern instances according to their potential severity.

Characterizing the Detected Anti-pattern Instances. In order to improve the readability of our report and help developers understand the detected anti-pattern instances faster, we provide a detailed breakdown of each detected anti-pattern instance in the report. Figure 6.1 shows an example detection report of the *nested transaction* anti-pattern. For each detected anti-pattern instance, the report shows the root transactional method (deleteUser) and the package to which it belongs. The report further shows that if any of the methods in the subsequently called methods contain a read or write to the DBMS (Recovered DB Access column in the report), and the transaction propagation of the root transactional method (**REQUIRED**). We also show the actual annotations that are declared in the code for the root transactional method (Annotation column), and whether the annotation is annotated at the method or class level. Finally, the report shows the nested transactional method (deleteUser), its propagation level, and the call path from the root transactional method to the nested transactional method. We provide similar breakdowns of each detected anti-pattern for other database access anti-patterns. In short, we find that by providing a detailed breakdown of each anti-pattern instance in the report, we can help developers understand the problem and uncover its root cause faster. Thus, developers can allocate the QA resources accordingly.

We provide a detailed breakdown of each detected anti-pattern instance in order to help developers understand the root cause of the problem faster and identify more important problems.

Learning From Developers. As discussed by Johnson *et al.* (2013), developers usually want more customizability of static anti-pattern detection tools or outputs. From our experience, we found that developers can tolerate a certain amount of false positives, but it is important for the anti-pattern detection tool to learn which detected patterns should not show up again in the detection report, based on developers' feedback. Integrating developers' feedback on what to do with a reported anti-pattern instance is useful for hiding detected database access anti-patterns that developers have already verified, or hiding anti-patterns that are less interesting (e.g., the anti-patterns have minor impact or the component with the detected anti-pattern is not a high priority component).

After discussing with developers, we implement a functionality in *DBChecker* to integrate developers' feedback to improve future reports. For example in Figure 6.1, developers can decide that all detected anti-pattern instances that have a **REQUIRED** transaction propagation should be hidden in future reports. Then, by clicking **REQUIRED** (under the column Propagation (ROOT)) in the first detected anti-pattern instance in the report, *all detected anti-pattern instances* that have the propagation level of **REQUIRED** would not appear in future reports.

On the other hand, if a developer decides to hide the detected anti-pattern instances according to the transactional method (e.g., deleteUser), then only that particular anti-pattern instance would be hidden. Developers can also use the text area (i.e., the form with a submit button in Figure 6.1) to see and to update their previous decisions.

In our experience, integrating developers' feedback on the detected anti-pattern instances can help developers prioritize their efforts.

C2: Giving Developers Rapid Feedback

Challenge Description. We found that it is difficult to ask every developer to run static anti-pattern detection tools in his/her own local environment. A similar challenge was previously encountered by Shen *et al.* (2011) while using other static anti-pattern detection tools. Setting up and running the tools may interrupt developers' common workflow. However, it is important to provide developers with prompt alerts about new anti-pattern instances in the code in a timely manner. If we only scan the code once a while, the tool may find a large number of newly introduced anti-pattern instances. Such a large number may reduce developers' motivation to inspect the detection results.

Challenge Impact. Based on our experience, if we only present the report to developers every once a while, we lower developers' attention and interest in the detected anti-pattern instances. Developers may forget about the details of the code that caused the anti-pattern instances, which makes it even more difficult to fix the detected anti-pattern instances. Moreover, since there may be new code that is dependent on the detected anti-pattern instances, fixing the detected anti-pattern instances may sometimes even require redesigning the APIs.

Solutions to the Giving Developers Rapid Feedback and Lessons Learned. In order to allocate resources to investigate detected database bug access patterns more efficiently, *DBChecker* is currently integrated in the Continuous Delivery process. Continuous Delivery (Chen, 2015a) is a common development process for ensuring the quality of the application, where development teams continuously generate products (newer versions of an application) that are reliable for releasing in short

cycles. For example, Facebook releases new version of its application into production twice a day, and Amazon makes changes to its production applications every 11.6 seconds on average[1].

To solve the above-mentioned issues, we host *DBChecker* in a cloud environment to scan the newest versions of the applications once a day, and generate a report of all the detected anti-pattern instances, as well as the new anti-pattern instances that are introduced since its last run. Since there is not usually a large amount of new code that is added since the last run, developers only need to examine a small number of newly introduced anti-pattern instances. In addition, developers do not need to worry about setting up and running *DBChecker* on their local environment.

Based on our experience, integrating our static anti-pattern detection tool in the Continuous Delivery process can help increase developers' interest in the detection results and allow prompt attention to the detected anti-pattern instances.

C3: Maintaining Developers' Interest in the Detection Results

Challenge Description. We found that developers may lose interest in the static anti-pattern detection results if the detected anti-pattern instances are not related to the components that are under active development, or if the detected anti-pattern instances are not related to the currently-faced development challenges. Namely, developers have goals in their development cycles, so they may focus more on their current goals first instead of allocating time to fix the detected anti-pattern instances that might not have an impact in the field yet.

[1]https://www.thoughtworks.com/insights/blog/case-continuous-delivery

Challenge Impact. Static anti-pattern detection tools are only useful if the developers are willing to investigate the detected anti-pattern instances and provide fixes. Thus, if developers lose interest in the tool, or do not trust the tool's output, the tool would provide no benefit to the developers.

Solutions to Maintaining Developers' Interest in the Detection Results and Lessons Learned. From our experience, we found that in order to increase developers' adoption and interest of static anti-pattern detection tools, it is important to have developers involved in tool development to some extent. In our previous chapter, we implemented our static anti-pattern detection tool and walked developers through the bugs that we found. The developers were not only interested in the anti-pattern instances that we detected, but they were also interested in how the tool was developed and whether the tool can be extended to detect other anti-patterns. Now we sometimes receive requests from developers to implement detectors for new anti-patterns that they see, and cannot be detected using existing tools such as FindBugs. We also found that developers have extremely high interest in reviewing the detected anti-pattern instances related to the new anti-patterns that they asked us to develop, since developers are still actively working on fixing those anti-patterns.

Involving developers in the development and discussion of the anti-pattern detection tool increases developers' interest and motivation to fix the detected anti-pattern instances.

C4: Communicating the Anti-patterns with Developers

Challenge Description. As applications become more complex, developers usually abstract SQL queries as method calls using various frameworks (e.g., Hibernate). However, since not all developers have deep understanding about the frameworks, we encountered some challenges when demonstrating the results of our newly implemented detection algorithm. Even after the tool is widely accepted, it is still very important to assign the right person to fix the anti-pattern to speed up the bug fixing process.

Challenge Impact. Developers may not take the static anti-pattern detection results seriously if they cannot understand the impact of the detected anti-pattern instances. Also, we found that sometimes developers may be unwilling to fix detected anti-pattern instances if they are not the one who introduced the detected anti-pattern instances.

Solutions to Communicating the Anti-patterns with Developers and Lessons Learned. We found that when demonstrating the tools to developers, it is important to educate them about the anti-patterns. Since it is impossible for every developer to understand all components of an application, some developers may not understand the impact and cause of some anti-patterns. Therefore, we had to find some key examples from the detected anti-pattern instances and demonstrate their impact. In short, we cannot simply give the detection reports without highlighting and demonstrating the reasons that the anti-pattern instances are detected, and the possible impact of the anti-pattern instances.

We hosted several “static anti-pattern detection result workshops” to advertise our tool to various development teams. We focused on explaining how the tool

works and the detection results, and letting developers know how the tool can be extended to help them detect other anti-patterns. One of the benefits of hosting such workshops is to learn more patterns from developers' experiences. In fact, we found most of the studied anti-patterns in this chapter through interacting with developers in the workshops.

We also found that some developers do not want to take the responsibility of fixing the detected anti-pattern instances. The reason can be that the developers are not familiar with the detected anti-pattern instances, or the developers think they are not the one who introduced the anti-patterns. Thus, it is important to determine an effective bug triaging mechanism or policy beforehand in order to react rapidly to the detected anti-pattern instances.

It is necessary to educate developers about the root causes and the possible impact of the detected anti-pattern instances to increase developers' awareness of the severity of these instances. In addition, an effective bug triaging policy is needed to react rapidly to the detected anti-pattern instances.

6.6 Database Access Anti-Patterns

6.6.1 The Need for Framework-Specific and Non-General Anti-Patterns

The detection algorithms that we use to detect the studied database access anti-patterns are straightforward, but knowing the anti-patterns in the first place requires extensive domain knowledge. As applications become more complex, developers start to leverage different frameworks and technologies during development.

There may be many new kinds of anti-patterns that are related to these frameworks, but since these anti-patterns are not available in most existing static anti-pattern detection tools, developers are left in the dark. As an example, a recent study (ZeroturnAround, 2014) found that there are three times more Java developers who use Hibernate (67.5%) than those who use JDBC (22%). However, although there are many JDBC-related patterns in the surveyed static anti-pattern tools, there is only one Hibernate-related anti-pattern (which is related to SQL injection) in Coverity. Therefore, finding anti-patterns that are more specific can further help improve application quality significantly. Due to the wide rise of frameworks throughout industry, in the following subsection, we discuss the framework-specific database access anti-patterns that we have seen in practice. Our goal of presenting these patterns is to give readers concrete examples of framework-specific anti-patterns. Hence, we hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language anti-patterns and associated detectors. Future studies should also consider detecting anti-patterns that may be more specific to certain frameworks but have a significant impact in most industrial applications.

6.6.2 New Anti-patterns

After our prototype (described in Chapter 5) (Chen *et al.*, 2014a) was adopted in practice, we received active feedback from developers, and they showed enormous interest in the detected anti-pattern instances. After active discussion and cooperation with the developers, we found five new database access anti-patterns. Instances of these anti-patterns have caused both functional and non-functional problems in

the applications, and some problems were difficult to capture. Unfortunately, current static anti-pattern detection tools such as FindBugs or PMD fail to capture most of these anti-patterns. As a result, we improved our *DBChecker* tool to further detect these five database access anti-patterns.

To provide more detailed information and breakdown of each anti-pattern, we discuss each anti-pattern using the following template:

Impact. Whether it is functional or non-functional (i.e. performance).

Description. A detailed description of the database access anti-pattern.

Example. An example of the anti-pattern.

Developer awareness. We search the database access anti-pattern on developer forums or blogs (e.g., Stack Overflow) to determine whether the anti-pattern affects other developers, or whether the anti-pattern is specific to our studied applications. We also summarize developers' discussions and thoughts.

Possible solutions. We discuss possible solutions to resolve the anti-pattern.

Detection approach. We briefly describe the implementation of our anti-pattern detection tool for detecting the anti-pattern.

Title: Nested Transaction

Impact. Non-functional.

Description. Developers may use the annotation **@Transactional** to execute a method and its subsequent method calls in a transaction. In addition to using the annotation directly, developers can also specify the properties for the transaction. The properties can be *REQUIRES_NEW* or *NOT_SUPPORTED* (as described in Section 6.4). When a method (e.g., method A) is annotated using **@Transactional**, and its subsequent method (e.g., method B) is annotated with properties such as *REQUIRES_NEW*, method B will be executed inside a new transaction. Then, the transaction in which method A resides will be suspended until B is finished. This is the intended behaviour of the properties; however, as the application becomes more complex, there may be other usages of method B that do not require method B to be executed in a separate transaction. In addition, the requirement of method A may be changed, and suspending the transaction may cause a transaction timeout. We also found that using the properties incorrectly can cause database deadlocks in practice. As a result, our tool detects and labels *Nested Transaction* as a warning, and developers are required to perform further inspection.

Example. As an example:

```
1 Class A{
2         @Transactional(timeout = 300ms)
3         public User updateUserById(int id) {
4                 ...
5                 notifyServer();
```

```
6                   ...
7           }
8 }
9 Class B{
10          @Transactional(REQUIRES_NEW)
11          public void notifyServer(){
12                  ...
13          }
14 }
```

Assuming that whenever the data of a user is updated, the server is notified about the event (e.g., for doing data analysis). In this example, there will be two transactions, one is created in updateUserById and another one is created in notifyServer. However, because the transaction property is *REQUIRES_NEW* in updateUserById, notifyServer would suspend the transaction in updateUserById until notifyServer is finished. Such transaction configuration may cause transaction timeouts (the timeout time is 300ms for updateUserById) and unnecessary transaction overhead (we may execute notifyServer asynchronously). Note that, if any subsequent method in notifySever also requires modifying or reading data in the User table, a deadlock may occur, because the suspended parent transaction is holding the lock but the second transaction is also trying to grant the lock.

Developer awareness. We found instances of developers discussing the potential problems of using *REQUIRES_NEW* incorrectly (Jedyk, 2014), such as blocking DBMS connection or deadlock. Thus, it is important to notify developers

about *nested transaction*, and manually investigate if the detected anti-patterns can cause potential problems.

Possible solutions. There are many possible solutions depending on the nature of the problem. For example, developers may remove the transaction property if the transaction is not needed, or they can execute the second transaction asynchronously if two transactions do not depend on each other. One may also refactor the code to place the annotation in other methods, or provide new APIs that have different transactional behaviours.

Detection approach. Our detection algorithm first constructs the call graph of the entire application, and records the annotation information for each method. Then, for each method that will be executed in a transaction, we traverse the method's call graph, and report a problem if any subsequent method in the call graph is annotated with **@Transactional** and have properties such as *REQUIRES_NEW*.

Title: Unexpected Transaction Behaviour

Impact. Functional.

Description. As the default behaviour of Spring's transaction management, the **@Transactional** annotation does not create a transaction if the annotated method is called within the same class (i.e., self-invocation). Hence, if developers call a method that is annotated with **@Transactional(REQUIRES_NEW)** within the same class, the method would not be executed in a new transaction, and the transaction would not be rolled back if errors occur.

Example. Consider the following example:

```
1  Class A{
2          @Transactional(timeout = 300ms)
3          public User updateUserById(int id) {
4                  ...
5                  notifyServer();
6          }
7          @Transactional(REQUIRES_NEW)
8          public void notifyServer(){
9                  ...
10         }
11 }
```

In this example, since both updateUserById and notifyServer are in the same class, notifyServer would not be executed in a separate transaction when called in updateUserById. Thus, the actual behaviour would be different from the developer's intention.

Developer awareness. There are many developers who are facing this problem and seek help online (StackOverflow, 2010b, 2014). Detecting such anti-pattern can be beneficial, because as shown in the Stack Overflow posts, developers may not be fully aware of the mechanism of how Spring manages transactions. As an application becomes more complex, it is difficult and time consuming to manually discover such problem in the code, or to notice such unexpected transaction behaviour during program execution.

Possible solution. To solve *Unexpected Transaction Behaviour*, one solution is to refactor the code so that methods annotated with **@Transactional(REQUIRES NEW)** are in a different class than the caller methods that are annotated with

@Transactional().

Detection approach. Our detection algorithm first constructs the call graph of the entire application, and records the annotation information for each method. Then, for each method that will be executed in a transaction, we traverse the method's call graph, and report it as an instance of the *Unexpected Transaction Behaviour* anti-pattern if any subsequent method in the call graph is annotated with **@Transactional(REQUIRES_NEW)** and both annotated methods are defined in the same class.

Title: Inconsistent Transaction Read-Write Level

Impact. Non-functional.

Description. Developers can specify a transaction annotation to be read-only when the annotated method (and its subsequent method calls) do not modify data in the DBMS. Setting a transaction to read-only provides a hint to the underlying Hibernate engine, and Hibernate may choose to open a read-only transaction, which has a smaller performance overhead compared to a read-write transaction. Note that even if the DBMS does not support read-only transaction, setting a transaction to read-only still has performance benefits when using Hibernate. Setting a transaction to read-only tells Hibernate not to automatically *flush* uncommitted changes to the DBMS. Flushes force Hibernate to synchronize the in-memory data to the DBMS (MIHALCEA, 2014), even though the transaction is not yet committed. Since read-only transactions do not modify the data in the DBMS, flushes are not necessary. Thus, setting the transaction to read-only can help improve performance even if the underlying

DBMS does not support read-only transactions. Despite the benefits of read-only transactions, sometimes developers may forget to set the transaction to read-only for methods with only read-access to the DBMS.

Example. Consider the following example:

```
1 @Transactional()
2 public User readUserById(int id) {
3         return session.find(User.Class, id)
4 }
```

In the above-mentioned example, the transaction read-write level is set to default, but the method only reads from the DBMS. Thus, ideally setting the transaction to read-only may help improve performance.

Developer awareness. In practice, we also see many developers who do not understand the difference between read-only and the default transaction level. For example, there are many posts on Stack Overflow asking the benefits of setting a transaction to read-only (StackOverflow, 2009, 2010a). Manually identifying all methods that have a mismatch between the transaction level and the database access can be a time-consuming task. Hence, automatically detecting such patterns can significantly reduce developers' effort.

Possible solution. The solution would require developers to change the annotated transaction to a read-only transaction for methods that do not modify data in the DBMS.

Detection approach. Our detection algorithm first recovers the call graph for the entire application, as well as the annotations that are associated with each method. Then, for each annotated method and its subsequent methods, we

traverse the call graph to see if there are any API calls that modify data in the DBMS. If there is only API calls that read data from the DBMS, and the method is not annotated with read-only, then we report the detected anti-pattern instance as a warning.

Title: Sequence Name Mismatch

Impact. Functional.

Description. In Hibernate, developers may choose which sequence object that a database entity class uses in the DBMS. Developers can add an annotation to an instance variable to specify the name of the sequence object that the variable uses. Sequence objects generate the next sequential number (e.g., primary key) when a new sequence object is created. However, there may be human errors that the name of the sequence object in the SQL schema file does not match with the name that is specified in the annotation in the code. In such cases, duplicated sequences may occur, and may cause duplicated primary key errors.

Example. Consider the following example:

```
1 Class User{
2         @Id
3         @SequenceGenerator(sequenceName=
4                 "user_seq")
5         @Column(name="user_id")
6         private int id;
7         ...
8 }
```

```
1 user_schema.sql
2 user_id BIGINT NOT NULL DEFAULT nextval ('user_id_seq')
```

In this example, we have a User class, which is mapped to the user table in the DBMS. The user id instance variable is mapped to the primary key column in the DBMS, and the name of the sequence object ("user_sql") is specified in the annotation **@SequenceGenerator**. However, in the user table schema file, we can see that the sequence name is "user_id_sql" and not "user_sql". Such sequence name mismatch may be caused by copy-and-paste error.

Developer awareness. Although there aren't many developer discussions online, copy-paste related bugs are common in practice (Li *et al.*, 2006). However, most existing static anti-pattern tools fail to detect *Sequence Name Mismatch*, since the error is caused by copy-paste errors between code and external SQL scripts, and most tools only consider source code files.

Possible solution. The solution is to use the same sequence name in both the annotation and in the SQL schema definition.

Detection approach. Our detection algorithms first scan all the annotations in the source code, and extract the sequence name in the annotation. Then, we scan all the SQL files, and look for mismatches between the sequence name that is specified in SQL and the sequence name that is specified in the annotation.

Title: Incorrect SQL Order

Impact. Functional.

Description. When using frameworks such as Hibernate, sometimes the order of the database access code and the generated SQL queries may not match. For

example, if developers try to first delete a user using a unique key and then reinsert the user with an updated key, the resulting SQL queries may be an update follows by a delete, which does not match developers' intended behaviour in the code. The reason of the mismatch is that Hibernate may reorder the SQL queries for some optimization, but such reordering causes unexpected problems to developers.

Example. Consider the following example:

```
1 group.getUserList().clear();
2 group.addUser(user);
3 update(group);
```

resulting SQL queries:

```
1 INSERT into User values ...
2 DELETE from User where ...
```

In this example, we are trying to first clear all the users in a group, and then add a new user to the group. However, the generated SQL queries first insert the new user and then delete users from the group. The order of the SQL queries is different from the order in the code. This may cause duplicate key problems, or result in unexpected outcomes.

Developer awareness. There are many discussions related to *incorrect SQL order* online (Forum, 2004; StackOverflow, 2015), and the problem often causes many unexpected functional errors.

Possible solution. A possible solution would be to force Hibernate to synchronize the in-memory data with the DBMS. Calling synchronization calls would allow the SQL queries to be executed in the correct order. However, manually

detecting where the problems are can be difficult, especially for large-scale applications.

Detection approach. To detect *incorrect SQL order*, we perform control flow analysis on the application source code. Our tool looks for Hibernate update and delete calls that will be executed together in a method under the same control flow. Our tool also ensures that the Hibernate update and delete calls will modify data in the same DBMS table to reduce the number of false positives.

6.7 Chapter Summary

In Chapter 5, we propose a static performance anti-pattern detection tool, and the tool is well received by our industrial partners. We then extend our tool and integrate the tool in practice. Thus, in this chapter, we provide an experience report on the challenges and lessons that we learned when adopting our extended static anti-pattern detection tool in practice. Our static anti-pattern detection tool focuses on detecting database access anti-patterns that existing static anti-pattern detection tools cannot detect. Since these database access anti-patterns are different from general code anti-patterns in many aspects, we discuss how we help developers prioritize detection results and what is needed to detect additional specialized anti-patterns. We also discuss how we implement the tool to increase developers' interest on the tool, and the importance of giving developers rapid feedback. Finally, we discuss five database access anti-patterns that we have observed in large-scale industrial applications over the past years. We believe that our findings can help researchers create static anti-pattern detection tools that have higher chances to be adopted in practice. We also highlight the need to create tools to detect more

framework-specific bugs, since most applications nowadays are leveraging frameworks instead of being built with basic programming constructs (the main focus of much of the current research in static anti-pattern detectors). However, static analysis has some well-known limitations (e.g., prone to false positives and lack runtime information so that some code paths may not be feasible in practice). Therefore, in order to propose a more comprehensive set of approaches in this thesis, in the next chapter, we propose an anti-pattern detection approach using dynamic analysis .

CHAPTER 7

Dynamically Detecting Redundant Data Anti-patterns

In the previous chapters, we propose an approach to statically detect performance anti-patterns in the source code. However, static analysis approaches are prone to false positives and lack runtime information. As a result, some of detected anti-pattern instances may not be valid in practice (i.e., no execution path can lead to the detected anti-pattern instances). Thus, in this chapter, we propose an automated approach, which we implement as a Java framework, to dynamically detect redundant data anti-patterns. We apply our framework on one enterprise and two open source applications. We find that redundant data anti-patterns exist in 87% of the exercised transactions. Due to the large number of detected redundant data anti-patterns, we propose an automated approach to assess the impact and prioritize the resolution efforts. Our performance assessment result shows that by resolving the redundant data anti-patterns, the application response time for the studied applications can be improved by an average of 17%.

An earlier version of this chapter is published at IEEE Transactions on Software Engineering (TSE), 2016. In Press. (Chen *et al.*, 2016d)

7.1 Introduction

SINCE ORM frameworks operate at the data-access level, ORM frameworks do not know how developers will use the data that is returned from the DBMS. Therefore, it is difficult for ORM frameworks to provide an optimal data retrieval approach for all applications that use ORM frameworks. Such non-optimal data retrieval can cause serious performance problems. We use the following example to demonstrate the problem. In some ORM frameworks (e.g., Hibernate, NHibernate, and Django), updating any column of a database entity object (object whose state is stored in a corresponding record in the database) would result in updating all the columns in the corresponding table. Consider the following code snippet:

```
1 // retrieve user data from DBMS
2 user.updateName("Peter");
3 // commit the transaction
4 ...
```

Even though other columns (e.g., address, phone number, and profile picture) were not modified by the code, the corresponding generated SQL query is:

> *update user set name='Peter', address='Waterloo', phone_number = '12345', profile_pic = 'binary data' where id=1;*

Such redundant data anti-patterns may bring significant performance overheads when, for example, the generated SQLs are constantly updating binary large objects (e.g., profile picture) or non-clustered indexed columns (e.g., assuming phone number is indexed) in a database table (Zaitsev *et al.*, 2008). The redundant data

anti-patterns may also cause a significant performance impact when the number of columns in a table is large (e.g., retrieving a large number of unused columns from the DBMS). Prior studies (Meurice and Cleve, 2014; Qiu *et al.*, 2013) have shown that the number of columns in a table can be very large in real-world applications (e.g., the tables in the OSCAR application have 30 columns *on average* (Meurice and Cleve, 2014)), and some applications may even have tables with more than 500 columns (StackoverFlow, 2016). Thus, detecting redundant data anti-patterns is helpful for large-scale real-world applications.

In fact, developers have shown that by optimizing ORM configurations and data retrieval, application performance can increase by as much as 10 folds (as discussed in Chapter 5 and 8). However, even though developers can change ORM code configurations to resolve different kinds of redundant data anti-patterns, due to the complexity of software applications, developers may not be able to detect such anti-patterns in the code, and thus may not proactively resolve the anti-patterns (Chen *et al.*, 2014a; Jovic *et al.*, 2011). Besides, there is no guarantee that every developer knows the impact of such anti-patterns.

In this chapter, we propose an approach for detecting redundant data anti-patterns in the code. We implemented the approach as a framework for detecting redundant data anti-patterns in Java-based ORM frameworks. Our framework is now being used by our industry partner to detect redundant data anti-patterns.

Redundant data or computation is a well-known cause for performance problems (Nistor *et al.*, 2013, 2015), and in this chapter, we focus on detecting database-related redundant data anti-patterns. Our approach consists of both static and dynamic analysis. We first apply static analysis on the source code to automatically

identify database access methods (i.e., methods that may access the data in the DBMS). Then, we use bytecode instrumentation on the application executables to obtain the code execution traces and the ORM generated SQL queries. We identify the needed database accesses by finding which database access methods are called during the application execution. We identify the requested database accesses by analyzing the ORM generated SQL queries. Finally, we discover instances of the redundant data anti-patterns by examining the data access mismatches between the needed database accesses and the requested database accesses, within and across transactions. Our hybrid (static and dynamic analysis) approach can minimize the inaccuracy of applying only data flow and pointer analysis on the code, and thus can provide developers a more complete picture of the root cause of the problems under different workloads.

We perform a case study on two open-source applications (Pet Clinic (PetClinic, 2016) and Broadleaf Commerce (Commerce, 2013)) and one large-scale Enterprise Application (EA). We find that redundant data anti-patterns exist in all of our exercised workloads. In addition, our statistical rigorous performance assessment shows that resolving redundant data anti-patterns can improve the application performance (i.e., response time) of the studied applications by 2–92%, depending on the workload. Our performance assessment approach can further help developers prioritize the efforts for resolving the redundant data anti-patterns according to their performance impact.

7.2 The Main Contributions of this Chapter

1. We survey the redundant data anti-patterns in popular ORM frameworks across four different programming languages, and we find that all surveyed frameworks share common problems.

2. We propose an automated approach to detect the redundant data anti-patterns in ORM frameworks using a hybrid approach (combining static and dynamic analysis), and we have implemented a Java-version to detect redundant data anti-patterns in Java applications.

3. Case studies on two open source applications and one enterprise application (EA) show that resolving redundant data anti-patterns can improve the application performance (i.e., response time) by up to 92% (with an average of 17%), when using MySQL as the DBMS and two separate computers, one for sending requests and one for hosting the DBMS. Our framework receives positive feedback from EA developers, and is now integrated into the performance testing process for the EA.

7.3 Related Work

Prior studies propose various approaches to detect different performance bugs through run-time indicators of such bugs. Nistor *et al.* (2013) propose a performance bug detection tool, which detects performance problems by finding similar memory-access patterns during application execution. Chis (2008) provide a tool to detect memory anti-patterns in Java heap dumps using a catalogue. Parsons and Murphy (2004)

present an approach for automatically detecting performance issues in enterprise applications that are developed using component-based frameworks. Parsons and Murphy (2004) detect performance issues by reconstructing the run-time design of the application using monitoring and analysis approaches.

Shen *et al.* (2015) propose a search-based approach for finding application performance bottlenecks by varying application input parameters. Xu *et al.* (2010b) introduce copy profiling, an approach that summarizes runtime activity in terms of chains of data copies, which are indicators of Java runtime bloat (i.e., many temporary objects executing relatively simple operations). Xiao *et al.* (2013) use different workflows to identify and predict workflow-dependent performance bottlenecks (i.e., performance bugs) in GUI applications. Xu *et al.* (2010a) introduce a run-time analysis to identify low-utility data structures whose costs are out of line with their gained benefits. Luo *et al.* (2015) propose an approach for finding performance bottlenecks in applications by analyzing program execution traces obtained from test executions.

7.4 Our Approach for Detecting Redundant Data Anti-patterns

The mapping between objects and database records can be complex, and usually contains some impedance mismatches (i.e., conceptual difference between relational databases and object-oriented programming). In addition, ORM frameworks do not know what data developers need and thus cannot optimize all the database operations automatically. In this section, we present our automated approach for

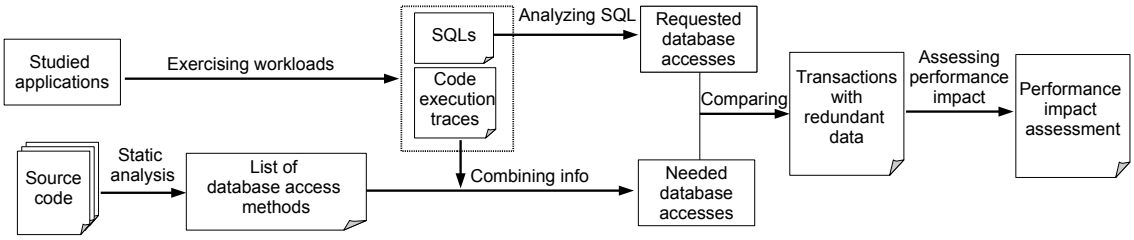


Figure 7.1: An overview of our approach for detecting and evaluating redundant data anti-patterns.

detecting the redundant data anti-pattern in the code due to ORM mapping. Note that our approach is applicable to other ORM frameworks in other languages (may require some framework-specific modifications).

7.4.1 Overview of Our Approach

Figure 7.1 shows an overview of our approach for detecting redundant data anti-patterns. We define the *needed database accesses* as how database access methods are called during application execution. We define the *requested database accesses* as the corresponding generated SQL queries during application execution. Our approach consists of three different phases. First, we use static source code analysis to automatically identify the database access methods (methods that read or modify instance variables that are mapped to database columns). Second, we leverage bytecode instrumentation to monitor and collect application execution traces. In particular, we collect the exercised database access methods (and the location of the call site of such methods) as well as the generated SQLs. Finally, we detect the redundant data anti-patterns by comparing the exercised database access methods and the SQLs. We explain the detail of each phase in the following subsections.

7.4.2 Identifying Needed Database Accesses

We use static code analysis to identify the mappings between database tables and the source code classes. We then perform static taint analysis on all the database instance variables (e.g., instance variables that are mapped to database columns) in database entity classes. Static taint analysis allows us to find all the methods along a method call graph that may read or modify a given variable. If a database instance variable is modified in a method, we consider the method as a *data-write method*. If a database instance variable is being read or returned in a method, we consider the method as a *data-read method*. For example, if a database entity class has an instance variable called *name*, which is mapped to a column in the database table, then the method getUserName(), which returns the variable *name*, is a data-read method. We also parse JPQL (Java Persistence Query Language, the standard SQL-like language for Java ORM frameworks) queries to keep track of which entity objects are retrieved/modified from the DBMS, similar to a prior approach proposed by Dasgupta *et al.* (2009). We focus on parsing the *FROM* and *UPDATE* clauses in JPQL queries.

To handle the situation where both superclass and subclass are database entity classes but they are mapped to different tables, we construct a class inheritance graph from the code. If a subclass is calling a database access method from its superclass, we use the result of the class inheritance graph to determine the columns that the subclass method is accessing.

```
<transaction>
        <functionCall>
                user.getUserName()
        </functionCall>
        <sql>
                select u.id, u.name, u.address,
                u.phone number from User u
                where u.id=1
        </sql>
</transaction>
```

Figure 7.2: An example of the exercised database access methods and generated SQL queries during a transaction.

7.4.3 Identifying Requested Database Accesses

We define the requested database accesses as the columns that are accessed in an SQL query. We develop an SQL query analyzer to analyze database access information in SQLs. Our analyzer leverages the SQL parser in FoundationDB (2015), which supports standard SQL92 syntax. We first transform an SQL query into an abstract syntax tree (AST), then we traverse the AST nodes and look for information such as columns that an SQL query is selecting from or updating to, and the tables that the SQL query is querying.

7.4.4 Finding Redundant Data

Since database accesses are wrapped in transactions (to assure the ACID property), we separate the accesses according to the transactions to which they belong. Figure 7.2 shows an example of the resulting data. In that XML snippet, the method call user.getUserName() (the needed data access) is translated to a select SQL (the requested data access) in a transaction.

We find redundant data anti-patterns at both the column and table level by comparing the needed and the requested database accesses within and across transactions. Since we know the database columns that a method is accessing, we compare the column reads and writes between the SQL queries and the database access methods. If a column that is being selected/updated in an SQL query has no corresponding method that reads/updates the column, then the transaction has a redundant data anti-pattern (e.g., in Figure 3.1 the Main.java only modifies user's name, but all columns are updated). In other words, an SQL query is selecting a column from the DBMS, but the column is not needed in the source code (similarly, the SQL query is updating a column but the column was not updated in the code). Note that after the static analysis step, we know the columns that a table (or database entity class) has. Thus, in the dynamic analysis step, our approach can tell us exactly which columns are not needed. In other words, our approach is able to find, for example, if a binary column is unnecessarily read from the DBMS, or if the SQL is constantly updating an unmodified but indexed column.

7.4.5 Performance Assessment

We propose an approach to automatically assess the performance impact of the redundant data anti-patterns. The performance assessment results can be used to prioritize performance optimization efforts. Since there may be different types of redundant data anti-patterns and each type may need to be assessed differently, we discuss our assessment approach in detail in Section 7.6.3, after discussing the types of redundant data anti-patterns that we discovered in Section 7.6.2.

Table 7.1: Statistics of the studied applications.

Application	Total lines of code (K)	No. of files	Max. No. of columns
Pet Clinic	3.3K	51	6
Broadleaf 3.0	206K	1,795	28
EA	$>$ 300K	$>$ 3,000	$>$ 50

7.5 Experimental Setup

In this Section, we discuss the studied applications and experimental setup.

7.5.1 Case Study Applications

We implement our approach as a framework, and apply the framework on two open-source applications (Pet Clinic (PetClinic, 2016) and Broadleaf Commerce (Commerce, 2013)) and one large-scale Enterprise Application (EA). Pet Clinic is an application developed by Spring (SpringSource, 2016), which provides a simple yet realistic design of a web application. Pet Clinic and its predecessor have been used in a number of performance-related studies (Chen *et al.*, 2014a; Grechanik *et al.*, 2012; Jiang *et al.*, 2008; Rohr *et al.*, 2008; van Hoorn *et al.*, 2008). Broadleaf (Commerce, 2013) is a large open source e-commerce application that is widely used in both non-commercial and commercial settings worldwide. EA is used by millions of users around the world on a daily basis, and supports a high level of concurrency control. Since we are not able to discuss the configuration details of EA due to a non-disclosure agreement (NDA), we also conduct our study on two open source applications. Table 7.1 shows the statistics of the three studied applications.

All of our studied applications are web applications that are implemented in Java. They all use Hibernate as their JPA implementation due to Hibernate's popularity (e.g., in 2013, 15% of the Java developer jobs requires the candidates to have Hibernate experience (Cheung *et al.*, 2013a)). The studied applications follow the typical "Model-View-Controller" design pattern (Krasner and Pope, 1988), and use Spring (SpringSource, 2016) to manage HTTP requests. We use MySQL as the DBMS in our experiment.

7.5.2 Experiments

Our approach and framework require dynamic analysis. However, since it is difficult to generate representative workloads (i.e., application use cases) for an application, we use the readily available performance test cases in the studied applications (i.e., Pet Clinic and EA) to obtain the execution traces. If the performance test cases are not present (i.e., Broadleaf), we use the integration test cases as an alternate choice. Both the performance and the integration test cases are designed to test different *features* in an application (i.e., use case testing). Both performance and integration test cases provide more realistic workloads and better test coverage (Binder, 2000). Table 7.2 shows the descriptions of the exercised test cases. Nevertheless, our approach can be adapted to deployed applications or to monitor real-world workloads for detecting redundant data anti-patterns in production.

We group the test execution traces according to the transactions to which they belong. Typically in database-related applications, a workload may contain one to many transactions. For example, a workload may contain *user login* and *user logout*, which may contain two transactions (one for each user operation).

7.6 Evaluation of Our Approach

In this section, we discuss how we implement our approach as a framework for evaluating our proposed approach, the redundant data anti-patterns that are discovered by our framework, and their performance assessment. We want to know if our approach can discover redundant data anti-patterns. If so, we want to also study what are the common redundant data anti-patterns and their prevalence in the studied applications. Finally, we assess the performance impact of the discovered redundant data anti-patterns.

7.6.1 Framework Implementation

To evaluate our approach, we implement our approach as a Java framework to detect redundant data anti-patterns in three studied JPA applications. We implement our static analysis tool for finding the needed database accesses using JDT (Eclipse, 2016b). We use AspectJ (Eclipse, 2016a) to perform bytecode instrumentation on the studied applications. We instrument all the database access methods in the database entity classes in order to monitor their executions. We also instrument the JDBC libraries in order to monitor the generated SQL queries, and we separate the needed and requested database accesses according to the transaction in which they belong (e.g., Figure 7.2).

7.6.2 Case Study Results

Using our framework, we are able to detect a large number of redundant data anti-patterns in the studied applications. In fact, on average 87% of the exercised

Table 7.2: Prevalence of the discovered redundant data anti-patterns in each test case. The detail of EA is not shown due to NDA.

Application	Test Case Description	Total No. of Trans.	Total No. of Trans. with Redundant Data	No. of Transactions with Redundant Data			
				Update All	Select All	Excessive Data	Per-Trans. Cache
Pet Clinic	Browsing & Editing	60	60 (100%)	6 (10%)	60 (100%)	50 (83%)	7 (12%)
Broadleaf	Phone Controller	807	805 (99%)	4 (0.5%)	805 (100%)	203 (25%)	202 (25%)
	Payment Info	813	611 (75%)	10 (1.6%)	611 (100%)	7 (1.1%)	200 (25%)
	Customer Addr.	611	609 (99%)	7 (1.1%)	607 (99%)	7 (1.1%)	203 (33%)
	Customer	604	602 (99%)	4 (0.7%)	602 (100%)	3 (0.5%)	200 (33%)
	Offer	419	201 (48%)	19 (9%)	19 (9%)	17 (9%)	201 (100%)
EA	Multiple Features	> 1000	$> 30\%$	3%	100%	0%	23%

Table 7.3: Overview of the redundant data anti-patterns that we discovered in our exercised workloads. *Trans.* column shows where the redundant data anti-pattern is discovered (i.e., within a transaction or across transactions).

Types	**Trans.**	**Description**
Update all	Within	Updating unmodified data
Select all	Within	Selecting unneeded data
Excessive data	Within	Selecting associated data but the data is not used
Per-trans cache	Across	Selecting unmodified data (caching problem)

transactions contain at least one redundant data anti-pattern. Our approach is able to find the redundant data anti-patterns in the code, but we are also interested in understanding what kinds of redundant data anti-patterns are there. Moreover, we use the discovered redundant data anti-patterns to illustrate the performance impact of the redundant data anti-patterns. However, other types of redundant data anti-patterns may still be discovered using our approach, and the types of the redundant data anti-patterns that we study here is by no means complete. In the following subsections, we first describe the type of redundant data anti-patterns that we discovered, then we discuss their prevalence in our studied applications.

Types of Redundant Data Anti-patterns

We perform a manual study on a statistically representative random sample of 344 transactions (to meet a confidence level of 95% with a confidence interval of 5% (Moore *et al.*, 2009)) in the exercised test cases that contain at least one redundant data anti-pattern (as shown in Table 7.2). We find that most instances of the redundant data anti-patterns can be grouped into four types, which we call:

update all, *select all*, *excessive data*, and *per-transaction cache* (other types of redundant data anti-patterns may still exist, and may be discovered using our approach). Table 7.3 shows an overview of the redundant data anti-patterns that we discovered in our exercised workloads.

Update all. When a developer updates some columns of a database entity object, *all* the database columns of the objects are updated (e.g., the example in Section 7.1). The redundant data anti-pattern is between the translations from objects to SQLs, where ORM simply updates all the database columns. This redundant data anti-pattern exists in some, but not all of the ORM frameworks. However, it can cause serious performance impact if not handle properly. There are many discussions on Stack Overflow regarding this type of redundant data anti-pattern (StackOverflow, 2016d). Developers complain about its performance impact when the number of columns or the size of some columns is large. For example, columns with binary data (e.g., pictures) would lead to a significant and unexpected overhead. In addition, this redundant data anti-pattern can cause significant performance impact when the generated SQLs are updating unmodified non-cluster indexed columns (Zaitsev *et al.*, 2008). In such cases, the index will need to be updated whenever the column is updated, even though the value of the column remains the same. Prior studies (Meurice and Cleve, 2014; Qiu *et al.*, 2013) have shown that the number of columns in a table can be very large in real-world applications (e.g., the tables in the OSCAR database have *on average* 30 columns (Meurice and Cleve, 2014)), and some applications may even have tables with more than 500 columns (StackoverFlow, 2016). Even in our studied applications, we find that some tables have more than 28, or even 50 columns (Table 7.1). Thus, this type of

redundant data anti-pattern may be more problematic in large-scale applications.

Select all. When selecting entity objects from the DBMS, ORM selects all the columns of an object, even though only a small number of columns are used in the source code. For example, if we only need a user's name, ORM will still select all the columns, such as profile picture, address, and phone number. Since ORM frameworks do not know what is the needed data in the code, ORM frameworks can only select all the columns.

We use the User class from Figure 3.1 as an example. Calling user.getName() ORM will generate the following SQL query:

> *select u.id, u.name, u.address, u.phone_number, u.profile_pic from User u where u.id=1.*

However, if we only need the user's name, selecting other columns may bring in undesirable data transmission or other performance overheads.

Developers also discuss the performance impact of this type of redundant data anti-pattern (McDonald, 2016; StackOverflow, 2016c). For example, developers are complaining that the size of some columns is too large, and retrieving them from the database causes performance issues (StackOverflow, 2016c). Even though most ORM frameworks provide a way for developers to customize the data fetch, developers still need to know how the data will be used in the code. The dynamic analysis part of our approach can discover which data is actually needed in the code (and can provide a much higher accuracy than using only static analysis), and thus can help developers configure ORM data retrieval.

Excessive Data. *Excessive data* is different from *select all* in all aspects, since this type of redundant data anti-pattern is caused by querying unnecessary entities from

other database tables. When using ORM frameworks, developers can specify relationships between entity classes, such as **@OneToMany**, **@OneToOne**, **@ManyToOne**, and **@ManyToMany**. ORM frameworks provide different optimization techniques for specifying how the associated entity objects should be fetched from the database. For example, a fetch type of ***EAGER*** means that retrieving the parent object (e.g., Group) will eagerly retrieve the child objects (e.g., User), regardless whether the child information is accessed in the source code.

If the relationship is ***EAGER***, then selecting Group will result in the following SQL:

> *select g.id, g.name, g.type, u.id, u.gid, u.name, u.address, u.phone_number, u.profile_pic from Group g left outer join User u on u.gid=g.id where g.id=1.*

If we only need the group information in the code, retrieving users along with the group causes undesirable performance overheads, especially when there are many users in the group.

ORM frameworks usually fetch the child objects using an SQL join, and such an operation can be very costly. Developers have shown that removing this type of *excessive data* anti-pattern can improve application performance significantly (Dubois, 2013). Different ORM frameworks provide different ways to resolve this redundant data anti-pattern, and our approach can provide guidance for developers on this type of anti-pattern.

Per-Transaction Cache. Our approach described in Section 7.4 also looks for redundant data anti-patterns across transactions (e.g., some data is repeatedly retrieved from the DBMS but the data is not modified). We find that the same SQLs

are being executed across transactions, with no or only a very little number of updates being called. *Per-transaction cache* is completely different from *one-by-one processing* studied in Chapter 5. *One-by-one processing* is caused by repeatedly sending similar queries with different parameters (e.g., in a loop) *within the same transaction*. *Per-transaction cache* is caused by non-optimal cache configuration: *different transactions* need to query the database for the same data even though the data is never modified. For example, consider the following SQLs:

```
update user set name='Peter', address='Waterloo', phone_number = '12345'
where id=1;
select u.id, u.name, u.address, u.phone number from User u where u.id=1;
...
select u.id, u.name, u.address, u.phone number from User u where u.id=1;
```

The first select is needed because the user data was previously updated by another SQL query (the same primary key in the *where* clause). The second select is not needed as the data is not changed. Most ORM frameworks provide cache mechanisms to reuse fetched data and to minimize database accesses (Section 7.7), but the cache configuration is never automatically optimized for different applications (Keith and Stafford, 2008). Thus, some ORM frameworks even turn the cache off by default. Developers are aware of the advantages of having a global cache shared among transactions (Sutherland and Clarke, 2016), but they may not proactively leverage the benefit of such a cache. We have seen cases in real-world large-scale applications where this redundant data anti-pattern causes the exact same SQL query to be executed millions of times in a short period of time, even though the retrieved entity objects are not modified. The cache configuration may

Table 7.4: Total number of SQLs and the number of duplicated selects in each test case.

Application	Test Case Description	Total No. of SQL queries	No. of Duplicate Selects
Pet Clinic	Browsing	32,921	29,882 (91%)
Broadleaf	Phone Controller	1,771	431 (24%)
	Payment Info	1,591	11 (0.7%)
	Customer Addr.	2,817	21 (0.7%)
	Customer	1,349	22 (1.6%)
	Offer	1,052	41 (3.9%)
EA	Multiple Features	$>>$ 10,000	$>$ 10%

be slightly different for different ORM frameworks, but our approach is able to give a detailed view on the overall data read and write. Thus, our approach discussed in this chapter can also assist developers with cache optimization.

Note that, although some developers are aware of the above-mentioned redundant data anti-patterns, it is not a common knowledge. Moreover, some developers may still forget to resolve the anti-pattern, as a redundant data anti-pattern may become more severe as an application ages.

Prevalence of Redundant Data Anti-patterns

Table 7.2 shows the prevalence of the redundant data anti-patterns in the executed test cases (a transaction may have more than one redundant data anti-pattern). Due to NDA, we cannot show the detail results for EA. However, we see that many transactions (>30%) have an instance of a redundant data anti-pattern in EA.

Most exercised transactions in BroadLeaf and Pet Clinic have at least one instance of redundant data anti-pattern (e.g., at least 75% of the transactions in the five test cases for BroadLeaf), and *select all* exists in almost every transaction. On

the other hand, *update all* does not occur in many transactions. The reason may be that the exercised test cases mostly read data from the database; while a smaller number of test cases write data to database. We also find that *excessive data* has a higher prevalence in Pet Clinic but lower prevalence in Broadleaf.

Since the *per-transaction cache* anti-pattern occurs across multiple transactions (caused by non-optimized cache configuration), we list the total number of SQLs and the number of duplicate selects (caused by *per-transaction cache*) in each test case (Table 7.4). We filter out the duplicated selects where the selected data is modified. We find that some test cases have a larger number of duplicate selects than others. In Pet Clinic, we find that the *per-transaction cache* anti-patterns are related to selecting the information about a pet's type (e.g., a bird, dog, or cat) and its visits to the clinic. Since Pet Clinic only allows certain pet types (i.e., six types), storing the types in the cache can reduce a large number of unnecessary selects. In addition, the visit information of a pet does not change often, so storing such information in the cache can further reduce unnecessary selects. In short, developers should configure the cache accordingly for different scenarios to resolve the *per-transaction cache* anti-pattern.

The four types of redundant data anti-patterns that are discovered by our approach have a high prevalence in our studied applications. We find that most transactions (on average 87%) contain at least one instance of our discovered anti-patterns, and on average 20% of the generated SQLs are duplicate selects (*per-transaction cache* anti-pattern).

7.6.3 Automated Performance Assessment

Since every ORM framework has different ways to resolve the redundant data anti-patterns, it is impossible to provide an automated ORM optimization for all applications. Yet, ORM optimization requires a great amount of effort and a deep understanding of the application workloads and design. Thus, to reduce developers' effort on resolving the redundant data anti-patterns, we propose a performance assessment approach to help developers prioritize their performance optimization efforts.

Assessing the Performance Impact of Redundant Data Anti-patterns

We follow a similar methodology as mentioned in Chapter 5.4.3 to automatically assess the performance impact of the redundant data anti-pattern. Note that our assessment approach is only for estimating the performance impact of redundant data anti-patterns in different workloads, and cannot completely fix the anti-patterns. Developers may wish to resolve these anti-patterns after further investigation. Below, we discuss the approaches that we use to assess each type of the discovered redundant data anti-patterns.

Assessing Update All and Select All. We use the needed database accesses and the requested database accesses collected during execution to assess *update all* and *select all* in the test cases. For each transaction, we remove the requested columns in an SQL if the columns are never used in the code. We implement an SQL transformation tool for such code transformation. We execute the SQLs before and after the transformation, and calculate the differences in response time after resolving the redundant data anti-pattern.

Table 7.5: Performance impact study by resolving the redundant data anti-patterns in each test case. Response time is measured in seconds at the client side. We mark the results in bold if resolving the redundant data anti-patterns has a statistically significant improvement. For response time differences, large/medium/small/trivial effect sizes are marked with L, M, S, and T, respectively.

Application	Test Case	Base	Update All		Select All		Excessive Data		Per-trans. Cache	
			resp. time	p-value	resp. time	p-value	resp. time	p-value	resp. time	p-value
Pet Clinic	Browsing & Editing	33.8±0.45	33.9±0.44 (0%)^{T}	0.85	**23.3±0.76 (-31%)**^{L}	**<<0.001**	**2.7±0.08 (-92%)**^{L}	**<<0.001**	**4.1±0.10 (-88%)**^{L}	**<<0.001**
Broadleaf	Phone Controller	14.0±0.69	**13.4±0.60 (-4%)**^{M}	**0.007**	13.9±0.65 (0%)^{T}	0.44	13.8±0.47 (-1%)^{S}	0.28	**13.0±0.56 (-7%)**^{L}	**<<0.001**
	Payment Info	18.7±2.6	**17.3±0.65 (-7%)**^{M}	**0.04**	18.1±0.88 (-3%)^{S}	0.37	18.3±1.0 (-2%)^{T}	0.58	18.0±0.74 (-4%)^{S}	0.33
	Customer Addr.	29.7±1.17	**28.4±0.58 (-4%)**^{M}	**<<0.001**	**28.7±0.45 (-3%)**^{M}	**0.001**	**29.0±0.68 (-2%)**^{S}	**0.05**	**28.8±0.54 (-3%)**^{S}	**0.008**
	Customer	13.7±0.63	**13.0±0.49 (-5%)**^{M}	**<<0.001**	**13.0±0.57 (-5%)**^{M}	**<<0.001**	**12.9±0.47 (-6%)**^{M}	**<<0.001**	**13.3±0.53 (-3%)**^{S}	**0.03**
	Offer	22.9±0.95	**21.3±1.05 (-7%)**^{L}	**<<0.001**	22.3±1.08 (-3%)^{S}	0.13	23.4±1.27 (+2%)^{S}	0.17	**21.9±0.87 (-4%)**^{M}	**0.002**
EA	Multiple Features	—	0%	>0.05	**> 30%**^{L}	**<<0.001**	—	—	**> 30%**^{L}	**<<0.001**

Assessing Excessive Data. Since we use static analysis to parse all ORM configurations, we know how the entity classes are associated. Then, for each transaction, we remove the eagerly fetched table in SQLs where the fetched data is not used in the code. We execute the SQLs before and after the transformation, and calculate the differences in response time after resolving the discrepancies.

Assessing Per-Transaction Cache. We analyze the SQLs to assess the impact of *per-transaction cache* in the test cases. We keep track of the modified database records by parsing the update, insert, and delete clauses in the SQLs. To improve the precision, we also parse the database schemas beforehand to obtain the primary and foreign keys of each table. Thus, we can better find SQLs that are modifying or selecting the same database record (i.e., according to the primary key or foreign key). We bypass an SQL select query if the queried data is not modified since the execution of the last *same SQL select*.

Results of Performance Impact Study

We first present a statistically rigorous approach for performance assessment. Then we present the results of our performance impact study.

Statistically rigorous performance assessment

Performance measurements suffer from variances during application execution, and such variances may lead to incorrect results (Georges *et al.*, 2007; Kalibera and Jones, 2013). As a result, we repeat the same steps as mentioned in Chapter 5.4.3. Namely, we repeat each test case 30 times, and use a *Student's t-test* and compute *Cohen's d* to obtain statistically rigorous results.

Results of Performance Impact Study

In the rest of this subsection, we present and discuss the results of our performance assessment. The experiments are conducted using MySQL as the DBMS and two separate computers, one for sending requests and one for hosting the DBMS (our assessment approach compares the performance between executing the original and the transformed SQLs). The response time is measured at the client side (computer that sends the requests). The two computers use Intel Core i5 as their CPU with 8G of RAM, and they reside in the same local area network (note that the performance overhead caused by data transfer may be bigger if the computers are on different networks).

Update All. Table 7.5 shows the assessed performance improvement after resolving each type of redundant data anti-pattern in each performance test case. For each test case, we report the total response time (in seconds) along with a confidence interval. In almost all test cases, resolving the *update all* anti-pattern gives a statistically significant performance improvement. We find that, by only updating the required columns, we can achieve a performance improvement of 4–7% with mostly medium to large effect sizes. Unlike select queries, which can be cached by the DBMS, update queries cannot be cached. Thus, reducing the number of update

queries to the DBMS may, in general, have a higher performance improvement. The only exception is Pet Clinic, because the test case is related to browsing, which only performs a very small number of updates (only six update SQL queries). EA also does not have a significant improvement after resolving *update all*.

As discussed in Section 7.6.2, the *update all* anti-pattern can cause a significant performance impact in many situations. In addition, many emerging cloud DBMSs implement the design of column-oriented data storage, where data is stored as sections of columns, instead of rows (Shang *et al.*, 2013). As a result, *update all* has a more significant performance impact on column-oriented DBMSs, since the DBMS needs to seek and update many columns at the same time for one update.

Select All. The *select all* anti-pattern causes a statistically significant performance impact in Pet Clinic, EA, and two test cases in Broadleaf (3–31% improvement) with varying effect sizes. Due to the nature of the Broadleaf test cases, some columns have null values, which reduce the overhead of data transmission. Thus, the effect of the *select all* anti-pattern is not as significant as the *update all* anti-pattern. In addition to what we discuss in Section 7.6.2, *select all* may also cause a higher performance impact in column-oriented DBMSs. When selecting many different columns from a column-oriented DBMS, the DBMS engine needs to seek for the columns in different data storage pools, which would significantly increase the time needed to retrieve data from the DBMS.

Excessive Data. We find that the *excessive data* anti-pattern has a high performance impact in Pet Clinic (92% performance improvement), but only 2–6% improvement in Broadleaf and 5% in EA with mostly non-trivial effect sizes. Since we know that the performance impact of the redundant data anti-pattern is highly dependent on

the exercised workloads, we are interested in knowing the reasons that cause the large differences. After a manual investigation, we find that the excessively selected table in Pet Clinic has a **@OneToMany** relationship. Namely, the transaction is selecting multiple associated excessive objects from the DBMS. On the other hand, most *excessive data* in Broadleaf has a **@ManyToOne** or **@OneToOne** relationship. Nevertheless, excessively retrieving single associated object (e.g., excessively retrieving the child object in a **@ManyToOne** or **@OneToOne** relationship) may still cause significant performance problems (StackOverflow, 2016e). For example, if the eagerly retrieved object contains large data (e.g., binary data), or the object has a deep inheritance relationship (e.g., the eagerly retrieved object also eagerly retrieves many other associated objects), the performance would also be impacted significantly.

Per-Transaction Cache. The *per transaction cache* anti-pattern has a statistically significant performance impact in 4 out of 5 test cases in Broadleaf with non-trivial effect sizes. We also see a large performance improvement in Pet Clinic, where resolving the *per-transaction cache* anti-pattern improves the performance by 88%. Resolving the *per-transaction cache* anti-pattern also improves the EA performance by 10% (with large effect sizes).

The performance impact of the *per-transaction cache* may be large if, for example, some frequently accessed read-only entity objects are stored in the DBMS and are not shared among transactions (Zaitsev *et al.*, 2008). These objects will be retrieved once for each transaction, and the performance overhead increases along with the number of transactions. Although the DBMS cache may be caching these

queries, there are still transmission and cache-lookup overheads. Our results suggest that the performance overheads can be minimized if developers use the ORM cache configuration in order to prevent ORM frameworks from retrieving the same data from the DBMS across transactions.

All of our detected instances of redundant data anti-patterns have a performance impact in all studied applications. Depending on the workloads, resolving the redundant data anti-patterns can improve the performance by up to 92% (17% on average). Our approach can automatically detect redundant data anti-patterns that have a statistically significant performance impact, and developers can use our approach to prioritize their performance optimization efforts.

7.7 A Survey on the Redundant Data Anti-patterns in Other ORM Frameworks

In previous sections, we apply our approach on the studied applications. We discover four types of redundant data anti-patterns, and we further illustrate their performance impact. However, since we only evaluate our approach on the studied applications, we do not know if the discovered redundant data anti-patterns also exist in other ORM frameworks. Thus, we conduct a survey on four other popular ORM frameworks across four programming languages, and study the existence of the discovered redundant data anti-patterns.

We study the documents on the ORM frameworks' official websites, and search for developer discussions about the redundant data anti-patterns. Table 7.6 shows the existence of the studied redundant data anti-patterns in the surveyed ORM frameworks under default configurations. Our studied applications use Hibernate

Table 7.6: Existence of the studied redundant data anti-patterns in the surveyed ORM frameworks (under default configurations).

Lang.	ORM Framework	Update all	Select all	Exce. Data	Per-trans. Cache
Java	Hibernate	Yes	Yes	Yes	Yes
Java	EclipseLink	No	Yes	Yes	Yes
C#	NHibernate	Yes	Yes	Yes	Yes
C#	Entity Framework	Yes	Yes	Yes	Yes
Python	Django	Yes	Yes	Yes	Yes
Ruby	ActiveRecord	No	Yes	Yes	Yes

as the Java ORM solution (i.e., one of the most popular implementations of JPA), and we further survey EclipseLink, NHibernate, Entity Framework, Django, and ActiveRecord. EclipseLink is another JPA implementation developed by the Eclipse Foundation. NHibernate is one of the most popular ORM solution for C#, Entity Framework is an ORM framework that is provided by Microsoft for C#, and Django is the most popular Python web framework, which comes with a default ORM framework. Finally, ActiveRecord is the default ORM for the most popular Ruby web framework, Ruby on Rails.

Update all. Most of the surveyed ORM frameworks have the *update all* anti-pattern, but the anti-pattern does not exist in EclipseLink and ActiveRecord (EclipseLink, 2016a; Rails, 2016). These two ORM frameworks keep track of which columns are modified and only update the modified columns. This is the design trade-off that the ORM developers made. The pros of the design decision is that this redundant data anti-pattern is handled by default. However, this will also introduce overheads such as tracking modifications and generating different SQLs for each update (StackOverflow, 2016b). All other surveyed ORM frameworks provide some

way for developers to customize the update to only update the modified columns (e.g., Hibernate supports a *dynamic-update* configuration). Although the actual fixes may be different, the idea on how to fix them is the same.

Select all. All of the surveyed ORM frameworks have the *select all* anti-pattern. The reason may be the ORM implementation difficulties, since ORM frameworks do not know how the retrieved data will be used in the application. Nevertheless, all the surveyed ORM frameworks provide some way to retrieve only the needed data from the DBMS (e.g., EclipseLink (2016b); StackOverflow (2016a)).

Excessive data. All of the surveyed ORM frameworks may have the *excessive data* anti-pattern. However, some ORM frameworks handle this anti-pattern differently. For example, the Django, NHibernate, Entity Framework, and ActiveRecord frameworks allow developers to specify the fetch type (e.g., ***EAGER*** v.s. ***LAZY***) for each data retrieval. Although Hibernate and EclipseLink require developers to set it at the class level, there are still APIs that can configure the fetch type for each data retrieval (Sutherland and Clarke, 2016).

Per-transaction cache. All of the surveyed ORM frameworks support some ways to share an object among transactions through caching. In the case of distributed applications, it is difficult to find a balance point between performance and stale data when using caches. Solving the anti-pattern will require developers to recover the entire workloads, and determine the tolerance level of stale data. Since our approach analyzes dynamic data, it can be used to help identify where and how to place the cache in order to optimize application performance.

7.8 Chapter Summary

In Chapter 5 and 6, we find that performance anti-patterns have a significant impact on application quality. Our proposed approaches in the previous chapters use static analysis to detect anti-patterns in the application; however, static analysis has its limitation, such as lack of runtime information and is prone to false positives.

In this chapter, we proposed an automated approach based on dynamic analysis to detect the redundant data anti-patterns in the code. Compared to using only static analysis, our approach is able to reduce the false positives by actually executing the application. Our dynamic analysis approach can also be applied in addition to the approaches mentioned in the previous chapters. Our dynamic analysis approach focuses more on how the queried data is used in the application; whereas our previous approaches focus more on detecting inefficient code patterns (i.e., do not consider how much data is actually transferred). In this chapter, we also proposed an automated approach for helping developers prioritize the efforts on fixing the redundant data anti-patterns.

We conducted a case study on two open source and one enterprise application to evaluate our hybrid approach. We found that, depending on the workflow, all the redundant data anti-patterns that are discussed in the chapter have statistically significant performance overheads, and developers are concerned about the impacts of these redundant data anti-patterns. Developers do not need to manually detect the redundant data anti-patterns in thousands of lines of code, and can leverage our approach to automatically detect and prioritize the effort to fix these redundant data anti-patterns.

CHAPTER 8

Automated ORM Cache Configuration Tuning

In Chapter 4, we find that developers do not usually tune performance-related configurations in ORM code, and yet such tuning is important for improving application performance. In Chapter 7, we propose an approach to identify potential non-optimal cache configurations in an application; however, as the workload of an application can be constantly changing (e.g., it may change on a daily or weekly basis), more is needed than simply pin-pointing the problems to developers. Thus, in this chapter, we propose CacheOptimizer, a lightweight approach that helps developers optimize the configuration of caching frameworks for web applications that are implemented using Hibernate. CacheOptimizer leverages readily-available web logs to create mappings between a workload and database accesses. Given the mappings, CacheOptimizer discovers the optimal cache configuration using coloured Petri nets, and automatically adds the appropriate cache configurations to the application. We evaluate CacheOptimizer on three open-source web applications. We find that i) CacheOptimizer improves the throughput by 27–138%; and ii) after considering both the memory cost and throughput improvement, CacheOptimizer still brings statistically significant gains (with mostly large effect sizes) in comparison to the application's default cache configuration and blindly enabling all possible caches.

An earlier version of this chapter is published at the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2016. Seattle, WA. Accepted. (Chen *et al.*, 2016a)

8.1 Introduction

APPLICATION-LEVEL caching frameworks, such as Ehcache (Terracotta, 2016) and Memcached (Memcached, 2016), are commonly used nowadays to speed up database accesses in large-scale web applications. Unlike traditional lower-level caches (e.g., hardware or web proxies) (Altinel *et al.*, 2003; Candan *et al.*, 2001; Chou and DeWitt, 1985; Johnson and Shasha, 1994), these application-level caching frameworks require developers to instruct them about what to cache, otherwise these frameworks are not able to provide any benefit to the application. Deciding what should be cached can be a very difficult and time-consuming task for developers, since they need to have in-depth knowledge of their applications and workload. For example, to decide that the results of a query should be cached, developers must first know that the query will be frequently executed, and that the fetched data is rarely modified. Furthermore, since caching frameworks are highly integrated with the application, these frameworks are configured in a very granular fashion – with cache API calls that are scattered throughout the code. Hence, developers must manually examine and decide on hundreds of caching decisions in their application. Even worse, a recent study finds that most database-related code is undocumented (Linares-Vasquez *et al.*, 2015), which makes manual configuration even harder.

Moreover, developers must continuously revisit their cache configuration as the workload of their application changes (Dar *et al.*, 1996). Outdated cache configurations may not provide as much performance improvement, and they might even lead to performance degradation. However, identifying workload changes is difficult in practice for large applications (Syer *et al.*, 2014; Zhao *et al.*, 2014). Even

knowing the workload changes, identifying performance bottlenecks is a difficult task (Fu *et al.*, 2012), and developers still need to spend great effort to understand the new workload and manually re-configure the caching framework.

In this chapter, we propose *CacheOptimizer*, a lightweight approach that automatically helps developers decide what should be cached (and also automatically places the cache configuration code) in web applications that are implemented using Hibernate (one of the most popular Java ORM frameworks) in order to optimize the configuration of caching frameworks. Using *CacheOptimizer*, developers can better manage the cost of their database accesses – greatly improving application performance (Bowman and Salem, 2005; Chen, 2015b; Chen *et al.*, 2014a; Cheung *et al.*, 2014; Ramachandra and Sudarshan, 2012).

CacheOptimizer first recovers the workload of a web application by mining the web server access logs. Such logs are typically readily-available even for large-scale applications that are deployed in production environments. *CacheOptimizer* further analyzes the source code statically to identify the database accesses that are associated with the recovered workloads. To identify detailed information about the recovered database accesses, such as the types of the access and the accessed data, *CacheOptimizer* leverages static taint analysis (Gollmann, 2011) to map the input variables of the web requests to the exact database accesses. Combining the recovered workload and the corresponding database accesses, *CacheOptimizer* models the workload, the database accesses, and the possible cache configurations as a coloured Petri net. By analyzing the Petri net, *CacheOptimizer* is able to determine an optimal cache configuration (i.e., given a workload, which objects or queries

should be cached by the caching frameworks). Finally, *CacheOptimizer* automatically adds the appropriate configuration calls to the caching framework API into the source code of the application.

We have implemented our approach as a prototype tool and evaluated it on three representative open-source database-centric web applications (Pet Clinic (PetClinic, 2016), Cloud Store (CloudScale, 2016), and OpenMRS (OpenMRS, 2016)) that are based on Hibernate (JBoss, 2016). Our approach has no overhead on the application that is deployed in production, and the static analysis step takes very little time (a few seconds to minutes). The choice of Hibernate is due to it being one of the most used Java platforms for database-centric applications in practice today (ZeroturnAround, 2014). However, our general idea of automatically configuring a caching framework should be extensible to other database abstraction technologies. We find that after applying *CacheOptimizer* to configure the caching frameworks on the three studied applications, we can improve the throughput of the *entire application* by 27–138%.

8.2 The Main Contributions of this Chapter

1. We propose an approach, called *CacheOptimizer*, which helps developers in automatically optimizing the configuration of caching frameworks for Hibernate-based web applications. *CacheOptimizer* does not require modification to existing applications for recovering the workload, and does not introduce extra performance overhead.

2. We find that the default cache configuration may not enable any cache or

may lead to sub-optimal performance, which shows that developers are often unaware of the optimal cache configuration. Our finding echoes with our observation in Chapter 4 and 7, where we find that developers rarely tune ORM performance configuration.

3. Compared to having no cache (*NoCache*), the default cache configurations (*DefaultCache*), and enabling all possible caches (*CacheAll*), *CacheOptimizer* provides better throughput improvement at a lower memory cost.

8.3 Related Work and Background

In this section, we discuss related work to *CacheOptimizer*. We focus on three closely related areas: software engineering research on software configuration, optimizing the performance of database-centric applications, and caching frameworks.

8.3.1 Software Configuration

Improving Software Configurations. Software configurations are essential for the proper and optimal operation of software applications. Several prior studies have proposed approaches to analyze the configurations of software applications. For example, Rabkin and Katz (2011b) use static analysis to extract the configuration options of an application, and infer the types of these configurations. Tianyin *et al.* (2015) conduct an empirical study on the configuration parameters in four open-source applications in order to help developers design the appropriate amount of

configurability for their application. Liu *et al.* (2015) focus on configuring client-slide browser caches for mobile devices. Grechanik *et al.* (2016) propose an approach to automatically learn behavioral models of an application, then use the model to guide developers on provisioning the application in the cloud.

Detecting and Fixing Software Configuration Problems. Rabkin and Katz (2011a) use data flow analysis to detect configuration-related functional errors. Zhang and Ernst (2013) propose a tool to identify the root causes of configuration errors. In another work, Zhang and Ernst (2014) propose an approach that helps developers configure an application such that the application's behaviour does not change as the application evolves. Chen *et al.* (2015) propose an analysis framework to automatically tune configurations to reduce energy consumption for web applications. Xiong *et al.* (2012) automatically generate fixes for configuration errors using a constraint-based approach.

Prior research on software configuration illustrates that optimizing configurations is a challenging task for developers. In this chapter, we propose *CacheOptimizer*, which particularly focuses on helping developers optimize the cache configurations to improve the performance of large-scale web applications.

8.3.2 Caching Frameworks

There are many prior studies on cache algorithms and frameworks. Many cache algorithms such as least recently used (LRU) (Johnson and Shasha, 1994), and most recently used (MRU) (Chou and DeWitt, 1985) are widely used in practice for scheduling lower-level caches. For example, such algorithms are used to improve

the performance of web applications by caching web pages through proxies (Candan *et al.*, 2001; Fan *et al.*, 2000). Most of these caching algorithms operate in an unsupervised dumb fashion, i.e., these low-level caching algorithms do not require any application-level knowledge to operate.

Many modern applications generate dynamic content, which may be highly variable and large in size, based on data in the DBMS. Therefore, many low-level cache frameworks are becoming less effective. Many recent caching frameworks cache database accesses at the application level (Fitzpatrick, 2004; Nishtala *et al.*, 2013). When using these application-level caching frameworks, developers have full control of what should be cached in an application. However, to leverage these caching frameworks effectively, they must be configured properly.

Unlike most prior studies, *CacheOptimizer* does not try to manage cache scheduling. Instead, *CacheOptimizer* is designed to help developers optimize the configuration of application-level caching frameworks, which must be configured correctly for developers to fully leverage their benefits.

8.4 CacheOptimizer

CacheOptimizer optimizes the configuration of caches that are associated with database accesses that occur for a given workload. Hence, our approach needs to recover the workload of an application then to identify which database access occurs within that particular workload. In the following subsections, we explain each step of the inner workings of *CacheOptimizer* in detail using a working example. The input of the working example shown in Figure 8.1 consists of two parts: 1) source code of

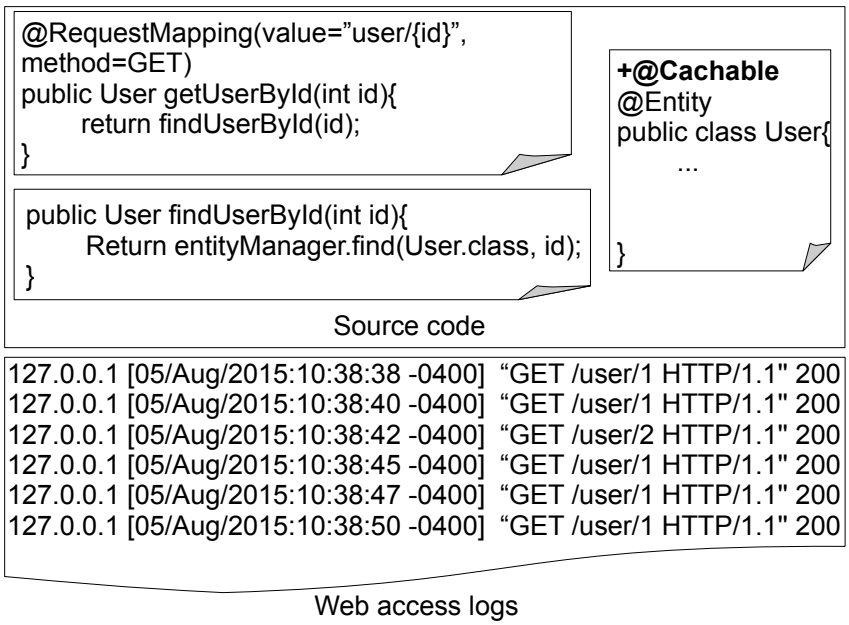


Figure 8.1: A working example of *CacheOptimizer*. The + sign in front of the @Cachable line indicates that the caching configuration is added by *CacheOptimizer*.

the application and 2) web access logs. Figure 8.2 shows an overview of *CacheOptimizer*. Note that we simulate the entire workload using a coloured petri net for all cache locations at once, but for the ease of understanding of our overall approach, we show the process for each cache location separately. Section 8.4.5 contains more details about our modeling approach using a coloured petri net.

8.4.1 Recovering Control and Data Flow Graphs

We first need to understand the calling and data flow relationships among methods, and determine which application-level methods are impacted by database caching (i.e., which methods eventually lead to a database access). We therefore extract the

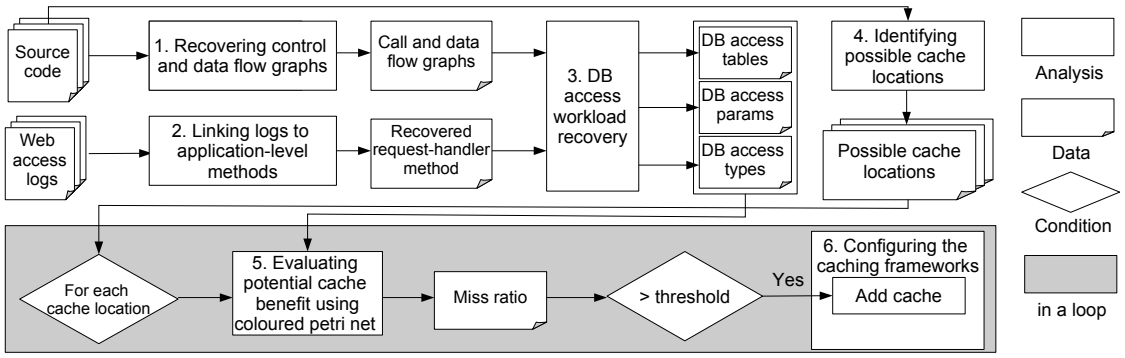


Figure 8.2: Overview of *CacheOptimizer*.

call and data flow graphs of the application by parsing the source code of the application using the Eclipse JDT. We opt to parse the source code instead of analyzing the binary since we need to locate the Hibernate annotations in the source code – such annotations are lost after compiling the source code to Java byte code.

We mark all Hibernate methods that access the DBMS (e.g., `query.execute()`) in the call and data flow graphs. Such methods are easy to identify since they are implemented in the same class (i.e., in the `EntityManager` and the `Query` class of Hibernate). Once such methods are marked, we are able uncover all the application-level methods that are likely to be impacted by optimizing the database cache.

In our working example, after generating the call and data flow graphs, and identifying the Hibernate database access methods, we would know that the method `getUserById` contains one database access, and the parameter is passed in through a web request.

8.4.2 Linking Logs to Application-Level Methods

We recover the workload of the application by mining its web access logs. We leverage web access logs because of the following reasons. First, since web access logs

are typically readily available without needing additional instrumentation, many database-centric applications rely on RESTFul web service (based on HTTP web requests) (Richardson and Ruby, 2008) to accept requests from users (Bloomberg, 2013). For example, large companies like IBM, Oracle, Facebook and Twitter all provide RESTFul APIs[1]. Second, unlike application logs, web access logs have a universal structure (the format of all log lines are the same) (Tomcat, 2016). Hence, compared to application logs, web access logs are easier to analyze and do not usually change as an application evolves (Shang *et al.*, 2011).

Web access logs may contain information such as the requestor's IP, timestamp, time taken to process the request, requested method (e.g. GET), and status of the response. An example web access log may look like:

127.0.0.1 [05/Aug/2015:10:38:38 -0400] 1202 "GET /user/1 HTTP/1.1" 200

This web access log shows that a request is sent from the local host at August 05, 2015 to get the information of the user whose ID is 1. The status of the response is 200, and the application took 1,202 milliseconds to respond to the request.

In order to know which application-level methods will be executed for each web request, we use static analysis to match the web access logs to application-level methods. *CacheOptimizer* parses the standard RESTful Web Services (JAX-RS) specifications in order to find the handler method for each web request (Oracle, 2015). An example of JAX-RS code is shown below:

[1]http://www.programmableweb.com/apis/directory

```
1 @RequestMapping(value = "/user/{id}", method=GET)
2 public User getUserById(int id) {
3         return findUserById(id);
4 }
```

In this example, based on the JAX-RS annotations, we know that all GET requests with the URL of form "/user/{id}" will be handled by the getUserById method.

For every line of web access log, *CacheOptimizer* looks for the corresponding method that handles that web request. After analyzing all the lines of web access logs, *CacheOptimizer* generates a list of methods (and their frequencies) that are executed during the run of the application.

In our working example, we map every line of web access log to a corresponding web request handling method, i.e., getUserById method.

8.4.3 Database Access Workload Recovery

We want to determine which database accesses are executed for the workload. Since application-level cache highly depends on the details of the database accesses, we need to recover the types of the database access (e.g., a query versus a select/insert/update/delete of a database entity object by id) and the data that is associated with the database access (e.g., accessed tables and parameters). Such detailed information of database accesses helps us in determining the optimal cache configurations. We first link each web access log to its request-handler method in code (as described in Section 8.4.2). Therefore, for each workload, we know the list of request-handler-methods that are executed (i.e., entry points into the application). Then, we conduct a call graph and static flow-insensitive interprocedural

taint analysis on each web-request-handler method, using the generated call and data flow graphs (as describe in Section 8.4.1).

Our algorithm for recovering the database access workload is shown in Algorithm 1. For each web-request-handler-method, we identify all possible database accesses by traversing all paths in the call graph, and recording the type of the database access. After recovering the database access, we traverse the data flow graph of each web-request-handler method to track the usage of the parameters that are passed in through the web requests. We want to see if the parameters are used for retrieving/modifying the data in the DBMS. Such information helps us better calculate the optimized cache configuration. For example, we would be able to count the number of times a database entity object is retrieved (e.g., according to the id that is specified in the web requests), or how many times a query is executed (e.g., according to the search term that is specified in the web request). For POST, PUT, and DELETE requests, we track the URL (e.g., POST /newUser/1) to which the request is sent, which usually specifies which object the request is updating. If there is no parameter specified, then we assume that the request may modify any of the objects to be conservative on our advice on enabling the cache.

In our working example, we recover a list of database accesses. All of the accesses read data from the User table. In five of the accesses, the parameter is 1 and in one of the accesses, the parameter is 2.

8.4.4 Identifying Possible Caching Locations

After our static analysis step, we recover the location of all the database access methods in the code, and the mapping between Java classes and tables in the DBMS.

Algorithm 1: Our algorithm for recovering database accesses.

Input: CG, DG, Mthd /* call graph, data flow graph, the request handler method */
Output: AccessInfo, Params /* accessed DB tables and DB func type (query or key-value lookup) and parameter of the request */

$AccessInfo \leftarrow \emptyset$; $Params \leftarrow \emptyset$;
/* Traverse the call graph from Mthd */
foreach *path* $\in$ *CG.findAllPathFrom(Mthd)* **do**
  foreach *call* $\in$ *path* **do**
    if *isDBCall(call)* **then**
      $AccessInfo \leftarrow AccessInfo \cup (getAccessedTable(call), getMthdType(call))$;
    end
  end
end
/* Track the usage of the input params */
foreach *param* $\in$ *Mthd.getParams()* **do**
  foreach *path* $\in$ *DG.findAllPathFrom(param)* **do**
    foreach *node* $\in$ *path* **do**
      $node \leftarrow pointToAnalysis(node)$
      if *usedInDBAccessCall(node)* **then**
        $Params \leftarrow Params \cup (dbAccessCall, node)$
      end
    end
  end
end

Namely, we obtain all potential locations for adding calls to the cache configuration APIs. Thus, if a query needs to be cached, we can easily find the methods in the code that execute the query. If we need to add object caches, we can easily find the class that maps to the object's corresponding table in the DBMS. In our example, we identify that the class User is a possible location to place an object cache. Note that our static analysis step is very fast (23–177 seconds on a machine with 16G RAM, and Intel i5 2.3GHz CPU) for our studied applications (Table 8.1), and is only required when deploying a new release. Thus, the execution time has minimal impact.

We use flow-insensitive static analysis approaches to identify possible caching locations, because it is extremely difficult to recover precise dynamic code execution paths without introducing additional overhead to the application (e.g., using instrumentation). During our static analysis step, if we choose to assign different probabilities to code branches, we may under-count or over-count reads and writes to the DBMS. Under-counting reads may result in failing to cache frequently read objects, which has little or no negative performance impact (i.e., the same as not adding a cache). However, under-counting writes may result in caching frequently modified objects and thus has significant negative effects on performance. In contrast, we choose a conservative approach by considering all possible code execution paths (over-counting) to avoid under-counting reads and writes. We may over-count reads and writes to the DBMS, but over-counting reads has minimal performance impact, since in such cases we would only place cache configuring APIs on objects that are rarely read from the DBMS; over-counting writes means that we may miss some objects that should have been cached, but will not affect the application performance (the same as adding no cache). Hence, our conservative choice by intentionally considering all possible code execution paths (over-counting) ensures that the caching suggestions would not have a negative performance impact after placing the suggested caches. Note that there may be some memory costs when turning on the cache (i.e., use more memory), and in RQ2 we evaluate the gain of our approach when considering such costs.

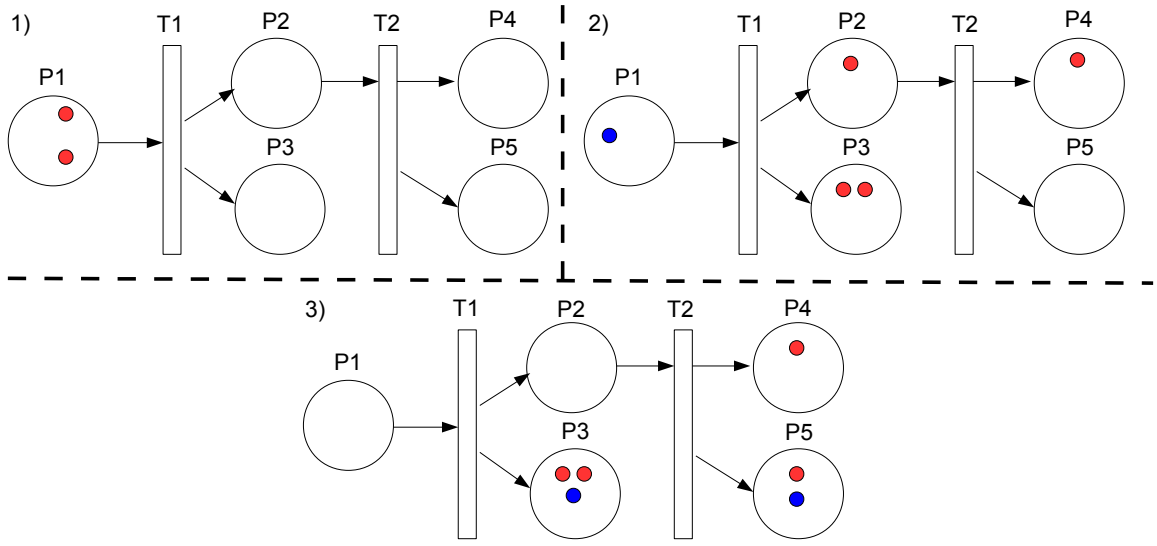


Figure 8.3: An example of modeling potential cache benefits using a coloured Petri net. A red token represents a read to a specific database entity object (e.g., *findUserById(1)*), and a blue token represents write to a specific database entity object (*updateUserById(1)*).

8.4.5 Evaluating Potential Cache Benefits Using Coloured Petri Net

After linking the logs to handler methods and recovering the database accesses, *CacheOptimizer* then calculates the potential benefits of placing a cache on each database access call. We use Petri nets (Peterson, 1981), a mathematical modeling language for distributed applications, to model the activity of caches such as cache renewal and invalidation. Petri nets allow us to model the interdependencies, so the reached caching decisions are global optimal, instead of focusing on top cache accesses (greedy). Petri nets model the transition of states in an application, and a net contains *places*, *transitions*, and *arcs*. Places represent conditions in the model, transitions represent events, and arcs represent the flow relations among places.

Formally, a Petri net N can be defined as:

$$N = (P, T, A) \text{ and } A \subset (P \times T) \cup (T \times P),$$

where P is the set of places, T is the set of transitions, and A is the set of arcs. Places may contain *tokens*, which represent the execution of the net. Any distributions of the tokens in the places of a net represent a set of *configurations*. A limitation of Petri nets is that there is no distinction between tokens. However, to use Petri nets to evaluate potential cache benefits, we need to model different data types (e.g., a Hibernate query versus an entity lookup by id) and values (e.g., query parameter). Thus, we use an extension of Petri nets, called coloured Petri net (CPN) (Jensen, 1997). In a CPN, tokens can have different values, and the values are represented using colours. Formally, a CPN can be defined as:

$$CPN = (P, T, A, \Sigma, C, N, E, G, I),$$

where P, T, and A are the same as in Petri nets. Σ represents the set of all possible colours (all possible tokens), C maps P to colours in Σ (e.g., specify the types of tokens that can be in a place), and N is a node function that maps A into $(P \times T) \cup (T \times P)$. E is the arc expression function, G is the guard function that maps each transition into guard expressions (e.g., boolean), and finally I represents an initialization function that maps each place to a multi-set of token colours.

In our CPN (shown in Figure 8.3), we define P to be the states of the data in the cache. P3 is a repository that stores the total number of database accesses, P4 stores the total number of cache hits, and P5 stores the number of invalidated caches. P2 is an intermediate place for determining whether the data would be cached or invalidated. We define T to be all database accesses that are recovered from the logs. We define Σ to distinguish the type of the database access call (e.g.,

read/write using ids or queries), and the parameters used for the access (obtained using Algorithm 1). Thus, our C defines that P4 can only have colours of database access calls that are reads, and P1, P2, P3, and P5 may contain all colours in Σ. The transition function on T1 always forwards the tokens in the initial place P1 to P2 and P3. There are two guard functions on T2, where one allows a token to be moved to P4 if there are two or more tokens of the same colour in P2 (i.e., multiple reads to the same data, so a cache hit), and another guard function makes sure that if there is a write in P2, all the same write tokens and the corresponding read tokens are moved to P5 (e.g., the cache is invalidated).

In our example (Figure 8.3), we let red tokens represent the database access call findUserById(1), and blue tokens represent updateUserById(1). In (1), there are two red tokens, and T1 is triggered, so the two red tokens are stored in P2 and P3. Since there are two red tokens in P2, T2 is triggered, and moves one red token to P4 (a cache hit). The resulting CPN is shown in (2). When a blue token appears in P1, T1 is triggered and moves the blue token to both P2 and P3. Since there is a blue token in P2, T2 is triggered, and we move both the red and blue token to P5 (cache invalidation). The final resulting Petri net is shown in (3). Note that T2 acts slightly different for tokens that represent query calls. When an object is updated, the query cache needs to retrieve the updated object from the DBMS to prevent a stale read. Thus, to model the behaviour, T2 would be triggered to move the query token to P5 from P2 if we see any token that represents a modification to the query table.

We use the recovered database accesses of the workload to execute the CPN. For *all tokens* that represent the database access to the same data (e.g., a read and

write to user by id 1), we examine their total counts in P3 and P4 to calculate the miss ratio (MR) of the cache. MR can be calculated as one minus the total number of cache hits in P4 divided by the total number of calls in P3 (i.e., number of hits divided by total number of requests). We choose MR because it is used in many prior studies to evaluate the effectiveness of caching (e.g., Fan *et al.* (2000); Iyer *et al.* (2002); Paul and Fei (2001); Zhou *et al.* (2001)). If MR is too high, caching the data would not give any benefit. For example, if a table is constantly updated, then we should not configure the caching framework to cache the data in that table. Thus, we define a threshold to decide whether a database access call should be cached. In our CPN, if MR is smaller than 35%, then we place the cache configuration code for the corresponding query (query cache) or table (object cache). Since object cache must be turned on to utilize query cache, we enable query cache only if the MR of the object cache is under the threshold. Such that, there would not exist conflicting decisions for object and query cache. We choose 35% to be more conservative on enabling caches so that we know the cached data would be invalidated less frequently (lower cache renewal cost). We also vary MR to 45% and do not see much performance difference compared to using an MR of 35%. However, future work should further investigate the impact of MR.

8.4.6 Configuring the Caching Frameworks

CacheOptimizer automatically adds the appropriate calls to the cache configuration API. Since the locations that require adding cache configuration APIs may be scattered across the code, *CacheOptimizer* helps developers reduce manual efforts by automatically adding these APIs to the appropriate locations. For example, if

the query that is executed by the request "/user/?query=Peter" should be cached, *CacheOptimizer* would automatically call the caching framework's API to cache the executed query in the corresponding handler method searchUserByName. In our example shown in Figure 8.1, the miss ratio of caching objects in the User class is 0.33, which is smaller than our threshold 0.35. *CacheOptimizer* automatically adds the *@Cachable* annotation to the source code to enable cache for the User class.

8.5 Evaluation

In this section, we present the evaluation of *CacheOptimizer*. We first discuss the applications that we use for our evaluation. Then we focus on two research questions: 1) what is the performance improvement after using *CacheOptimizer*; and 2) what is the gain of *CacheOptimizer* when considering the cost of such caches.

Experimental Setup. We evaluate *CacheOptimizer* on three open-source web applications: Pet Clinic (PetClinic, 2016), Cloud Store (CloudScale, 2016), and OpenMRS (OpenMRS, 2016). Table 8.1 shows the detailed information of these three applications. All three applications use Hibernate as the underlying framework to access database, and use MySQL as the DBMS. We use Tomcat as our web server, and use Ehcache as our underlying caching framework. Pet Clinic, which is developed by Spring (SpringSource, 2016), aims to provide a simple yet realistic design of a web application. Cloud Store is a web-based e-commerce application, which is developed mainly for performance testing and benchmarking. Cloud Store follows the TPC-W performance benchmark standard (TPCW, 2016). Finally, OpenMRS is large-scale open-source medical record application that is used worldwide. OpenMRS supports both web-based interfaces and RESTFul services.

Table 8.1: Statistics of the studied applications.

	Total lines of code	Number of Java files
Pet Clinic	3.8K	51
Cloud Store	35K	193
OpenMRS	3.8M	1,890

We use one machine each for the DBMS (8G RAM, Xeon 2.67GHz CPU), web server (16G RAM, Intel i5 2.3GHz), and JMeter load driver (12G RAM, Intel Quad 2.67GHz). The three machines are all connected on the same network. We use performance test suites to exercise these applications when evaluating *CacheOptimizer*. Performance test suites aim to mimic the real-life usage of the application and ensure that all of the common features are covered during the test (Binder, 2000). Thus, for our evaluation, performance test suites are a more appropriate and logical choice over using functional tests. We use developer written tests for Pet Clinic (Dubois, 2013), and work with BlackBerry developers on creating the test cases for the other applications. For Cloud Store, we create test cases to cover searching, browsing, adding items to shopping carts, and checking out. For OpenMRS, we use its RESTFul APIs to create test cases that are composed of searching (by patient, concept, encounter, and observation etc), and editing/adding/retrieving patient information. We also add randomness to our test cases to better simulate real-world workloads. For example, we add some randomness to ensure that some customers may checkout, and some may not. We use, for our performance tests, the MySQL backup files that are provided by Cloud Store and OpenMRS developers. The backup file for Cloud Store contains data for over 5K patients and 500K observations. The backup file for Cloud Store contains about 300K customer data and 10K items.

RQ1: What is the performance improvement after using CacheOptimizer?

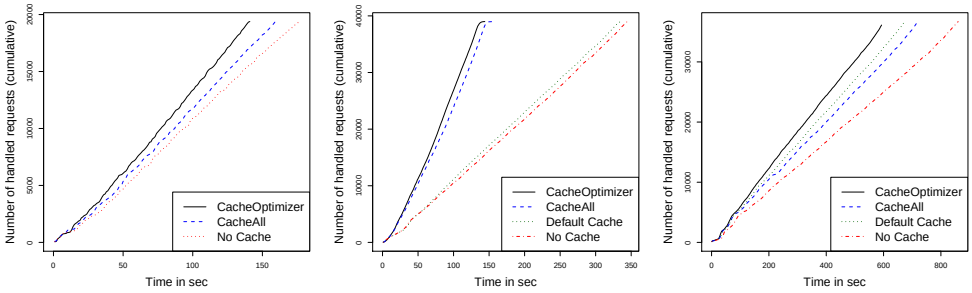
Motivation. In this RQ, we want to examine how well the performance of the studied database-centric web applications can be improved when using *CacheOptimizer* to configure the caching framework.

Approach. We run the three studied applications using the performance test suites under four different sets of cache configurations: 1) without any cache configuration (*NoCache*), 2) with default cache configuration (*DefaultCache*, cache configurations that are already in the code, which indicates what developers think should be cached), 3) with enabling all possible caches (*CacheAll*), and 4) with configurations that are added by *CacheOptimizer*. We compare the performance of the applications when configured using these four different sets of cache configurations. We work with performance testing experts from BlackBerry to ensure that our evaluation steps are appropriate, accurate, and realistic. We use throughput to measure the performance. The throughput is measured by calculating the number of requests per second throughout the performance test. A higher throughput shows the effectiveness of the cache configuration, as more requests can be processed within the same period of time. There may exist many possible locations to place the calls to the cache configuration APIs. Hence, configuring the caching framework may require extensive and scattered code changes, which can be a challenging and time-consuming task. Therefore, to study the effectiveness of *CacheOptimizer* and how it helps developers, we also compare the number of cache configurations that are added by *CacheOptimizer* relative to the total number of all possible caching configurations that could be added, and the number of cache configurations that exist in *DefaultCache*.

Table 8.2: Performance improvement (throughput) against *NoCache* after applying different cache configurations.

	Throughput			
	NoCache	*CacheOptimizer*	*CacheAll*	*DefaultCache*
Pet Clinic	98.7	125.1 (+27%)	108.4 (+10%)	—
Cloud Store	110.7	263.4 (+138%)	249.3 (+125%)	114.7 (+4%)
OpenMRS	21.3	30.8 (+45%)	25.5 (+20%)	27.7 (+30%)

Results. *CacheOptimizer* outperforms *DefaultCache* and *CacheAll* in terms of application performance improvement. Table 8.2 shows the performance improvement of the applications under four sets of configurations. We use *NoCache* as a baseline, and calculate the throughput improvement after applying *CacheOptimizer*, *CacheAll*, and *DefaultCache*. The default cache configuration of Pet Clinic does not enable any cache. Therefore, we only show the performance improvement of *DefaultCache* for Cloud Store and OpenMRS. Using *CacheOptimizer*, we see a throughput improvement of 27%, 138% and 45% for Pet Clinic, Cloud Store and OpenMRS, respectively. The throughput improvement of applying *CacheOptimizer* is always higher than that of *DefaultCache* and *CacheAll* for all the studied applications. Figure 8.4 further shows the cumulative throughput overtime. We can see that for the three studied applications, the throughput is about the same at the beginning regardless of us adding cache or not. However, as more requests are received, the benefit of caching becomes more significant. The reason may be that initially when the test starts, the data is not present in the cache. *CacheOptimizer* is able to discover the more temporal localities (reuse of data) in the workload and help developers configure the application-level cache more optimally (Jin and Bestavros, 2001). Therefore, as more requests are processed, frequently accessed



(a) Pet Clinic. (b) Cloud Store. (c) OpenMRS.

Figure 8.4: Number of handled requests overtime (cumulative).

Table 8.3: Total number of possible places to add cache in the code, and the number of location that are enabled by *CacheOptimizer* and that exist in the *DefaultCache*.

	Object Cache			Query Cache		
	Total	*CacheOptimizer*	*DefaultCache*	Total	*CacheOptimizer*	*DefaultCache*
Pet Clinic	11	6 (55%)	0	4	3 (75%)	0
Cloud Store	33	2 (6%)	10 (30%)	24	9 (38%)	1 (4%)
OpenMRS	112	16 (14%)	7 (6%)	229	2 (0.9%)	0

data is then cached, which significantly reduces the overhead of future accesses. We see a trend that the longer the test runs, the more benefit we get from adding cache configuration code using *CacheOptimizer*. We also observe that the performance of Cloud Store with *DefaultCache* is close to the performance with no cache. Such an observation shows in some instances, developers do not have a good knowledge of optimizing cache configuration in their own application.

***CacheOptimizer* enables a small number of caches to improve performance.** *CacheOptimizer* can help developers change cache configurations quickly without manually investigating a large number of possible cache locations (e.g., object cache or query). Table 8.3 shows the total number of possible locations to place calls to object and query cache APIs in the studied applications.

We also show the number of *CacheOptimizer* enabled caches, and the number of *DefaultCache* enabled caches. *CacheOptimizer* suggests adding object cache configuration APIs to a fraction (6–55%) of the total number of possible cache locations. In OpenMRS and Cloud Store, where there are more Hibernate queries, *CacheOptimizer* is able to improve performance by enabling 0.9% and 38% of all the possible caches, respectively. For the object cache of Cloud Store, *CacheOptimizer* even suggests enabling a smaller number of caches than *DefaultCache*. For large applications like OpenMRS with 112 possible object caches and 229 possible query caches, manually identifying the optimized cache configuration is time-consuming and may not even be possible.

Discussion. In our evaluation of *CacheOptimizer*, we observe a larger improvement in Cloud Store. After manual investigation, we find that *CacheOptimizer* caches the query results that contain large binary data, e.g., pictures. Since the sizes of pictures are often larger, caching them significantly reduces the network transfer time, and thus results in a large performance improvement. We see less improvement when using *DefaultCache*, because most database access calls are done through queries, while the default cache configurations of Cloud Store are mostly for object cache (see Table 8.3). Thus, enabling only object caches does not help improve performance. In OpenMRS, both *CacheOptimizer* and *DefaultCache* cache some database entity objects that are not often changed. However, *CacheOptimizer* is able to identify more object caches and queries that should be cached to further improve performance. We also see that the overhead of *CacheAll* causes OpenMRS to run slower when compared to *DefaultCache*. In Pet Clinic, we find that caching the owner information significantly improves the performance of searches. Moreover, since the

number of vets in the clinic is often unchanged, caching the vet information also speeds up the application.

Adding cache configuration code, as suggested by CacheOptimizer , improves throughput by 27–138%, which is higher than using the default cache configuration or enabling all possible caches. The sub-optimal performance of DefaultCache shows that developers have limited knowledge of adding cache configuration.

RQ2: What is the gain of CacheOptimizer when considering the cost of such caches?

Motivation. In the previous RQ, we see that *CacheOptimizer* helps improve application throughput significantly. However, caching may also bring some memory overhead to the application, since we need to store cached objects in the memory. As a result, in this RQ, we want to evaluate *CacheOptimizer* -suggested cache configuration when considering both the cost (increase in memory usage) and the benefit (improvement in throughput).

Approach. In order to evaluate *CacheOptimizer* when considering both benefit and cost, we define the *gain* of applying a configuration as:

$$Gain(c) = Benefit(c) - Cost(c), \tag{8.1}$$

where c is the cache configuration, $Gain(c)$ is the *gain* of applying c, while $Benefit(c)$ and $Cost(c)$ measure the benefit and the cost, respectively, of applying c. In our case study, we measure the throughput improvement in order to quantify the benefit of caching, and we measure the memory overhead in order to quantify the cost of caching. We use the throughput and memory usage when no cache is added to the application as a baseline. Thus, $Benefit(c)$ and $Cost(c)$ are defined as follows:

$$Benefit(c) = TP(c) - TP(no\ cache), \tag{8.2}$$

$$Cost(c) = MemUsage(c) - MemUsage(no\ cache), \tag{8.3}$$

where $TP(c)$ is the average number of processed requests per second with cache configuration c, and $MemUsage(c)$ is the average memory usage with cache configuration c.

Since the throughput improvement and the memory overhead are not in the same scale, the calculated *gain* by Equation 8.1 may be biased. Therefore, we linearly transform both $Benefit(c)$ and $Cost(c)$ into the same scale by applying min-max normalization, which is defined as follows:

$$x' = \frac{(x - x_{min})}{(x_{max} - x_{min})}, \tag{8.4}$$

where x and x' are the values of the metric before and after normalization, respectively; while x_{max} and x_{min} are the maximum and the minimum values of the metric, respectively. We note that if one wants to compare the *gain* of applying multiple configurations, the maximum and the minimum values of the metric are

calculated by considering all the values of the metrics across the different configurations, including having no cache. For example, if one would like to compare the *gain* of applying *CacheOptimizer* and *CacheAll*, $throughput_{max}$ is the maximum throughput of applying *CacheOptimizer*, *CacheAll*, and *NoCache*. After the transformation, the *gain* represents the ratio between the increase in throughput and in memory usage. We then compare the ratio across different cache configurations.

To evaluate *CacheOptimizer*, in this RQ, we compare the *gain* of applying *CacheOptimizer*, *CacheAll*, and *DefaultCache* against *NoCache*. The larger the *gain*, the better the cache configuration. If the *gain* is larger than 0, the cache configuration is better than using *NoCache*. In order to understand the *gain* of leveraging cache configuration throughout the performance tests, we split each performance test into different periods. Since a performance test with different cache configurations runs for a different length of time (see Figure 8.4), we split each test by each thousand of completed requests. For each period, we calculate the *gain* of applying *CacheOptimizer*, *CacheAll*, and *DefaultCache*.

We study whether there is a statistically significant difference in *gain*, between applying *CacheOptimizer* and *CacheAll*, and between applying *CacheOptimizer* and *DefaultCache*. To do this we use the Mann-Whitney U test (Wilcoxon rank-sum test) (Moore *et al.*, 2009), as the *gain* may be highly skewed. Since the Mann-Whitney U test is a non-parametric test, it does not have any assumptions on the distribution. A p-value smaller than 0.05 indicates that the difference is statistically significant. We also calculate the effect sizes in order to quantify the differences in *gain* between applying *CacheOptimizer* and *CacheAll*, and between applying *CacheOptimizer* and *DefaultCache*. Unlike the Mann-Whitney U test, which only tells

Table 8.4: Comparing the *gain* of the application under three different configurations: *CacheOptimizer*, *CacheAll*, and *DefaultCache*

	gain(*CacheOptimizer*) > *gain*(*CacheAll*)?		*gain*(*CacheOptimizer*) > *gain*(*DefaultCache*)?	
	p-value	Cliff's d	p-value	Cliff's d
Pet Clinic	$<< 0.001$	0.81 (large)	—	—
Cloud Store	< 0.01	0.32 (small)	$<< 0.001$	0.61 (large)
OpenMRS	$<< 0.001$	0.95 (large)	$<< 0.001$	0.95 (large)

us whether the difference between the two distributions is statistically significant, the effect size quantifies the difference between the two distributions. Since reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, the p-value are likely to be small even if the difference is trivial) (Cohen, 1977; Kampenes *et al.*, 2007), we use Cliff's d to quantify the effect size (Cliff, 1993). We choose Cliff's d because it is a non-parametric effect size measure, which does not have any assumption of the underlying distribution. Cliff's d is defined as:

$$\textit{Cliff's } d = \frac{\#(x_i > x_j) - \#(x_i < x_j)}{m * n}, \tag{8.5}$$

where $\#$ is defined the number of times, and the two distributions are of the size m and n with items x_i and x_j, respecitvely. We use the following thresholds for Cliff's d (Cliff, 1993):

$$\text{effect size} = \begin{cases} \text{trivial} & \text{if } \textit{Cliff's } d < 0.147 \\ \text{small} & \text{if } 0.147 \leq \textit{Cliff's } d < 0.33 \\ \text{medium} & \text{if } 0.33 \leq \textit{Cliff's } d < 0.474 \\ \text{large} & \text{if } 0.474 \leq \textit{Cliff's } d \end{cases}$$

Results. *CacheOptimizer* outperforms *DefaultCache* and *CacheAll* when considering the cost of cache. Table 8.4 shows the result of our Mann-Whitney U test and Cliff's d value when comparing the *gain* of applying *CacheOptimizer* with that of *CacheAll* and with *DefaultCache*. We find that in all three studied applications, the *gain* of *CacheOptimizer* is better than the *gain* of *CacheAll* and *DefaultCache* (statistically significant). The p-values are all smaller than 0.05. We also find that the effect sizes of comparing *CacheOptimizer* with *CacheAll* on *gain* are large for Pet Clinic (0.81) and OpenMRS (0.95). The only exception is Cloud Store, where the Cliff's d value indicates that the effect of *gain* is small (0.32) when comparing *CacheOptimizer* with *CacheAll*. On the other hand, when compared to *DefaultCache*, *CacheOptimizer* has a large effect size for both Cloud Store and OpenMRS.

Discussion. We investigate the memory overhead of applying *CacheOptimizer*, *CacheAll*, and *DefaultCache*. We use the Mann-Whitney U test and measure effect sizes using Cliff's d to compare the memory usage between applying *CacheOptimizer* and the memory usage of having no cache, *CacheAll*, and *DefaultCache*, respectively. The memory usage of applying *CacheOptimizer* and having no cache is statistically indistinguishable for Pet Clinic and OpenMRS; while for Cloud Store, applying *CacheOptimizer* has statistically significantly more memory usage than having no cache with a large effect size (0.78). This may explain why we see larger throughput improvement in Cloud Store. For OpenMRS, the memory usage of applying *CacheOptimizer* and *DefaultCache*is statistically indistinguishable. Finally, when comparing *CacheOptimizer* with *CacheAll*, we find that for Pet Clinic and Cloud Store, the difference in memory usage is statistically indistinguishable; while for OpenMRS, *CacheOptimizer* uses statistically significantly less memory than

CacheAll (p-value < 0.01) with an effect size of 0.61 (large effect). Nevertheless, after considering both the improvement and cost, *CacheOptimizer* out-performs all other cache configurations.

When considering both the benefit (throughput improvement) and cost (memory overhead), the gain of applying CacheOptimizer is statistically significantly higher than CacheAll and DefaultCache.

8.6 Threats to Validity

External Validity. We only evaluated *CacheOptimizer* on three applications, hence our findings may not generalize to other applications. We choose the studied applications with various sizes across different domains to make our results more generalizable. However, evaluating *CacheOptimizer* on other applications would further show the generalizability of our approach. We implement *CacheOptimizer* specifically for Hibernate-based web applications. However, the approach in *CacheOptimizer* should be applicable to applications using different object-relational mapping frameworks or other database abstraction technologies. For example, our approach for recovering the database accesses from logs may also be used by non-Hibernate based applications. With minor modifications (e.g., changes are needed to the definitions of the tokens and transition functions in the coloured Petri net), *CacheOptimizer* can be leveraged to improve cache configurations of other applications.

Construct Validity. The performance benefits of caching highly depends on the workloads. Thus, we use performance tests to evaluate *CacheOptimizer*. It is possible that the workload from the performance tests may not be representative enough

for field workload. However, *CacheOptimizer* does not depend on a particular workload, nor do we have any assumption on the workload when conducting our experiments. *CacheOptimizer* is able to analyze any given workload and find the optimal cache configuration for different workloads. If the workload changes greatly and the cache configuration is no longer optimal, *CacheOptimizer* can saves developers' time and effort by automatically finding a new optimal cache configuration. For example, developers can feed their field workloads on a weekly or monthly basis, and *CacheOptimizer* would help developers optimize the configuration of their caching frameworks. To maximize the benefit of caching, our approach aims to "overfit" the cache configurations to a particular workload. Thus, similar to other caching algorithms or techniques, our approach will not work if the workload does not contain any repetitive reads from the DBMS.

Our approach for recovering the database access. Prior research leverages control flow graphs to recover the executed code paths using logs (Zhao *et al.*, 2014). We do not leverage control flow graphs to recover the database accesses from web access logs for two reasons. First, as a basic design principal of RESTFul web services, typically one web-request-handing method maps to one or very few database accesses (IBM, 2015; Richardson and Ruby, 2008). Second, although leveraging control flows may give us richer information about each request, it is impossible to know which branch would be executed based on web access logs. Heuristics may be used to calculate the possibility of taking different code paths. However, placing the cache incorrectly can even cause performance degradation. Thus, to be conservative when enabling caching and to ensure that *CacheOptimizer* would always help improve performance, we consider all possible database access calls. Our

overestimation ensures that *CacheOptimizer* would not cache data that has a high likelihood of being frequently modified, so the *CacheOptimizer* added cache configurations should not negatively impact the performance. Future research should consider the use of adding control flow information for optimizing the cache configurations.

Cache concurrency level. There are different cache concurrency levels, such as read-only and read/write. In this chapter, we only consider the default level, which is read/write. Read/write cache concurrency strategy is a safer choice if the application needs to update cached data. However, considering other cache concurrency levels may further improve performance. For example, read-only caches may perform better than read/write cache if the cached data is never changed. A possible extension for our work could add cache concurrency level information to *CacheOptimizer* when trying to optimize cache configuration.

Distributed cache environment. Cache scheduling is a challenging problem in a distributed environment due to cache concurrency management. Most application-level caching frameworks provide different algorithms or mechanisms to handle such issues. Since the goal of *CacheOptimizer* is to instruct these caching frameworks on what to cache, we rely on the underlying caching frameworks for cache concurrency management. However, the benefit of using *CacheOptimizer* may not be as pronounced in a distributed environment.

8.7 Chapter Summary

Modern large-scale database-centric web-based cloud applications usually leverage different application-level caching frameworks, such as Ehcache and Memcached,

to improve performance. However, these caching frameworks are different from traditional lower-level caching frameworks, because developers need to instruct these application-level caching frameworks about what to cache. Otherwise these caching frameworks are not able to provide any benefit to the application.

In Chapter 4 and 7, we find that developers rarely tune ORM cache configuration. Therefore, in this chapter, we propose *CacheOptimizer*, a lightweight approach that helps developers in deciding what should be cached in order to utilize such application-level caching frameworks for Hibernate-based web applications. Compared to our proposed approach in Chapter 7, *CacheOptimizer* can automatically tune ORM cache configuration, and do not need developers to manually examine/modify every possible caching location detected by our anti-pattern detection framework. *CacheOptimizer* combines static analysis on source code and logs to recover the database accesses, and uses a coloured Petri net to model the most effective caching configuration for a particular workload. Finally, *CacheOptimizer* automatically updates the code with the appropriate calls to the caching framework API. We evaluate *CacheOptimizer* on three open source web applications (Pet Clinic, Cloud Store, and OpenMRS), and we find that *CacheOptimizer* improves the throughput of *the entire application* by 27–138% (higher compared to *CacheAll* and *DefaultCache*), and the increased memory usage is smaller than the applications' default cache configuration and turning on all caches. The sub-optimal performance of the default cache configurations highlights the need for automated techniques to assist developers in optimizing the cache configuration of database-centric web-based cloud applications.

CHAPTER 9

Conclusion and Future Work

THIS chapter summarizes the main ideas that are presented in this thesis. In addition, we propose future work to leverage program analysis to help improve application performance.

Managing the data consistency between the source code and the DBMS is a difficult task, especially for complex large-scale applications. Developers nowadays usually use different frameworks to abstract database accesses. For example, Object-Relational Mapping (ORM) frameworks are very popular among developers. Using ORM frameworks, changes in the object states are propagated automatically to the corresponding records in the database. These abstraction frameworks significantly reduce the amount of code that developers need to write (Barry and Stanienda, 1998; Leavitt, 2000); however, due to the black box nature of such abstraction layers, developers may not fully understand the behaviours of the framework-generated SQL queries, which may result in performance problems. Therefore, we believe that in order to help improve the performance of database-centric applications, it is important to help developers write better database access code. To evaluate our hypothesis, we propose different approaches to find problems in the database access code using program analysis, with a focus on ORM code due to its popularity. Our results show that our approaches are valuable to software engineering practitioners, and highlight the need in industry for supports from the research community.

9.1 Thesis Contributions

This thesis aims to first understand then propose approaches to help developers write better and more efficient database access code. The contributions of the thesis

also fill up gaps in the state-of-the-art, as shown in our literature review in Chapter 2. In our literature review, we find that most prior studies only focus on the database world (i.e., SQL queries). However, most database accesses are now abstracted as method calls, so developers do not need to manually write SQL queries anymore. Below, we highlight the main contributions of this thesis.

1. **Maintenance Activities of ORM Code.** We conduct an exploratory study on the maintenance activities of ORM code, and we find that using ORM comes with its own cost. We find that ORM cannot completely encapsulate database accesses in objects or abstract the underlying database technology, thus may cause ORM code changes to be more scattered; ii) ORM code changes are more frequent than regular code, but there is a lack of tools that help developers verify ORM code at compilation time; iii) changes to ORM code are more commonly due to performance or security reasons; however, traditional static code analyzers rarely capture the peculiarities of ORM code in order to detect such problems. Our study highlights the need for approaches that help software engineering practitioners improve the performance of database-centric applications.

2. **Statically Detecting ORM Performance Anti-patterns.** To help developers, we first propose an automated framework to detect ORM performance anti-patterns. Our framework automatically flags performance anti-patterns in the source code. Furthermore, as there could be hundreds or even thousands of instances of anti-patterns, our framework provides support to prioritize performance bug fixes using a statistically rigorous performance assessment.

We successfully evaluated our framework on one open source and one large-scale industrial applications. Our case studies show that our framework can detect new and known real-world instances of performance anti-patterns and that fixing the detected anti-patterns can improve the application response time by up to 69%.

3. **Adopting Anti-pattern Detection Framework in Practice.** Our anti-pattern detection framework receives positive feedback from industry, and the framework is extended for detecting additional anti-patterns. We discuss the challenges that we encountered and the day-to-day lessons that we learned during the integration of our framework into the development processes. Since most applications nowadays are leveraging frameworks, we also provide a detailed discussion of five additional framework-specific database access anti-patterns patterns that we found. We hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language anti-patterns and associated detectors.

4. **Dynamically Detecting Redundant Data Anti-patterns.** Since static analysis has its own limitation (e.g., may have false positives, and the results may not reflect real problems during execution), we also propose a dynamic approach to locate redundant data anti-patterns. Such anti-patterns are difficult to detect using static analysis. We find that redundant data anti-patterns exist in 87% of the exercised transactions in our studied applications. Due to the large number of detected instances of the redundant data anti-patterns, we propose an automated approach to assess the impact and prioritize the resolution efforts. Our performance assessment result shows that by resolving the

detected instances of redundant data anti-patterns, the application response time for the studied applications can be improved by an average of 17%.

5. **Automated ORM Cache Configuration Tuning.** As we find in our exploratory study, developers usually do not tune performance-related ORM configurations. Thus, to further help improve the performance of these database-centric applications, we propose an approach to help developers automatically find the optimal configurations. We propose CacheOptimizer, a lightweight approach that helps developers optimize the configuration of caching frameworks for cloud-based web applications that are implemented using Hibernate (one of the most popular ORM frameworks). CacheOptimizer leverages readily-available web logs to create mappings between a workload and database accesses. Given the mappings, CacheOptimizer discovers the optimal cache configuration using coloured Petri nets, and automatically adds the appropriate cache configurations to the application. We evaluate CacheOptimizer on three open-source web applications. We find that i) CacheOptimizer improves the throughput by 27–138%; and ii) after considering both the memory cost and throughput improvement, CacheOptimizer still brings statistically significant gains (with mostly large effect sizes) in comparison to the applications default cache configuration and blindly enabling all possible caches.

9.2 Future Research Directions

The approaches that are proposed in thesis show promising results on improving the performance of database-centric applications. However, there are still many challenges and improvements that can be addressed in future research.

9.2.1 Leveraging Operational Data to Improve Application Performance

As shown in Chapter 8, we can leverage the information in logs to help find optimal cache configurations in applications. Future studies can follow the approach and use the log-recovered information to identify performance hotspots in the code, and prioritize bug fixing efforts.

9.2.2 Recommending API Usage and Coding Structure

Many of the studied performance problems in this thesis are related to incorrect usage of the ORM APIs, or incorrect coding structure (e.g., the one-by-one processing anti-pattern shown in Chapter 5). Therefore, it will be beneficial if we can automatically recommend and instruct developers to write more performant code during software development.

9.2.3 Documenting and Detecting Additional Performance Anti-patterns

We propose several approaches to detect database-related performance anti-patterns in this thesis, but this list is far from complete. Moreover, for different kinds of applications (e.g., mobile) may have different anti-patterns due to the nature of the application. Future studies need to examine how performance anti-patterns differ across platforms and languages.

9.2.4 Finding Mismatches between DBMS and ORM Configurations

Although ORM aims to abstract the underlying DBMS, there can still be some mismatches in between the ORM configurations and the DBMS. For example, as shown in Chapter 6, there can be mismatches in ORM configuration and database schema. Therefore, future studies should study the mismatches between DBMS and ORM configurations, given the high frequency of ORM code changes.

Bibliography

Ackermann, H., Reichenbach, C., Mller, C., and Smaragdakis, Y. (2015). A backend extension mechanism for PQL/Java with free run-time optimisation. In *Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 111–130. Springer Berlin Heidelberg.

Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., and Reinwald, B. (2003). Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 718–729.

Anton, A. and Potts, C. (2003). Functional paleontology: the evolution of user-visible system services. *IEEE Transactions on Software Engineering*, **29**(2), 151–166.

Apache (2016). Apache OpenJPA. http://openjpa.apache.org/. Last accessed March 8 2016.

Arnold, M., Hind, M., and Ryder, B. G. (2002). Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 111–129.

Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y. (2007). Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8.

Baltopoulos, I., Borgstrm, J., and Gordon, A. (2011). Maintaining database integrity with refinement types. In *ECOOP 2011 Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 484–509. Springer Berlin Heidelberg.

Barry, D. and Stanienda, T. (1998). Solving the Java object storage problem. *Computer*, **31**(11), 33–40.

Bauer, C. and King, G. (2005). *Hibernate in Action*. In Action. Manning.

Binder, R. (2000). *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley.

Bloom, R. (2013). log4jdbc. https://code.google.com/p/log4jdbc/. Last accessed April 14 2013.

Bloomberg, J. (2013). *The Agile Architecture Revolution: How Cloud Computing, REST-Based SOA, and Mobile Computing Are Changing Enterprise IT*. Wiley.

Boslaugh, S. and Watters, P. (2008). *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). O'Reilly Media.

Bowman, I. T. and Salem, K. (2005). Optimization of query streams using semantic prefetching. *ACM Transactions on Database Systems*, **30**(4), 1056–1101.

Candan, K. S., Li, W.-S., Luo, Q., Hsiung, W.-P., and Agrawal, D. (2001). Enabling dynamic content caching for database-driven web sites. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 532–543.

Cao, W. and Shasha, D. (2013). Appsleuth: A tool for database tuning at the application level. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 589–600.

Chaudhuri, S., Narasayya, V., and Syamala, M. (2007). Bridging the application and DBMS profiling divide for database application developers. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1252–1262. VLDB Endowment.

Chavan, M., Guravannavar, R., Ramachandra, K., and Sudarshan, S. (2011a). Dbridge: A program rewrite tool for set-oriented query execution. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1284–1287.

Chavan, M., Guravannavar, R., Ramachandra, K., and Sudarshan, S. (2011b). Program transformations for asynchronous query submission. In *Proceedings of the*

2011 IEEE 27th International Conference on Data Engineering, ICDE '11, pages 375–386.

Chen, F., Grundy, J., Schneider, J.-G., Yang, Y., and He, Q. (2015). StressCloud: A tool for analysing performance and energy consumption of cloud applications. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, ICSE '15, pages 721–724.

Chen, L. (2015a). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, **32**(2), 50–54.

Chen, T.-H. (2015b). Improving the quality of large-scale database-centric software systems by analyzing database access code. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, ICDE'15, pages 245–249.

Chen, T.-H., Weiyi, S., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2014a). Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 1001–1012.

Chen, T.-H., Nagappan, M., Shihab, E., and Hassan, A. E. (2014b). An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91.

Chen, T.-H., Weiyi, S., Hassan, A. E., Nasser, M., and Flora, P. (2016a). CacheOptimizer: Helping developers configure caching frameworks for Hibernate-based database-centric web applications. In *Proceedings of the 24th International Symposium on the Foundations of Software Engineering*, FSE.

Chen, T.-H., Weiyi, S., Hassan, A. E., Nasser, M., and Flora, P. (2016b). Detecting problems in the database access code of large scale systems - an industrial experience report. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 71–80.

Chen, T.-H., Weiyi, S., Yang, J., Hassan, A. E., Nasser, Godfrey, M. W., Mohamed, and Flora, P. (2016c). 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.

Chen, T.-H., Weiyi, S., Jiang, h. M., Hassan, A. E., Nasser, M., and Flora, P. (2016d). 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*.

Cheung, A., Madden, S., Arden, O., and Myers, A. C. (2012a). Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, **5**(11), 1471–1482.

Cheung, A., Solar-Lezama, A., and Madden, S. (2012b). Inferring SQL queries using program synthesis. *CoRR*.

Cheung, A., Solar-Lezama, A., and Madden, S. (2013a). Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14.

Cheung, A., Arden, O., Madden, S., and Myers, A. C. (2013b). Speeding up database

applications with pyxis. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 969–972.

Cheung, A., Arden, O., Madden, S., Solar-Lezama, A., and Myers, A. C. (2013c). Statusquo: Making familiar abstractions perform using program analysis. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*.

Cheung, A., Madden, S., and Solar-Lezama, A. (2014). Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 931–942.

Chis, A. E. (2008). Automatic detection of memory anti-patterns. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 925–926.

Chou, H.-T. and DeWitt, D. J. (1985). An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 127–141.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, **114**(3), 494–509.

CloudScale (2016). Cloud store. http://www.cloudscale-project.eu/. Last accessed March 8 2016.

Cohen, J. (1977). *Statistical power analysis for the behavioral sciences*. Academic Press.

Commerce, B. (2013). Broadleaf commerce. http://www.broadleafcommerce.org/.

Community, J. (2016). Hibernate. http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch20.html#performance-fetching. Last accessed March 8 2016.

Cook, W. R. and Rai, S. (2005). Safe query objects: Statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 97–106.

Cook, W. R. and Wiedermann, B. (2011). Remote batch invocation for SQL databases. In *Proceedings of the 13th International Symposium on Database Programming Languages (DBPL)*, pages 1–6.

Cooper, E. (2009). The script-writers dream: How to write great sql in your own language, and be sure it will succeed. In *Database Programming Languages*, volume 5708 of *Lecture Notes in Computer Science*, pages 36–51.

Coverity (2016). Coverity code advisor. http://www.coverity.com/. Last accessed March 8 2016.

Curino, C. A., Moon, H. J., and Zaniolo, C. (2008a). Graceful database schema evolution: The prism workbench. *Proceedings of the VLDB Endowment*, **1**(1), 761–772.

Curino, C. A., Tanca, L., Moon, H. J., and Zaniolo, C. (2008b). Schema evolution in Wikipedia: toward a web information system benchmark. In *Proceedings of the International Conference on Enterprise Information Systems*, ICEIS 2008, pages 323–332.

Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., and Ziauddin, M. (2004). Automatic sql tuning in oracle 10g. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, VLDB '04, pages 1098–1109. VLDB Endowment.

Dar, S., Franklin, M. J., Jónsson, B. T., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 330–341.

Dasgupta, A., Narasayya, V., and Syamala, M. (2009). A static analysis framework for database applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1403–1414.

Dubois, J. (2013). Improving the performance of the spring-petclinic sample application. http://blog.ippon.fr/2013/03/14/improving-the-performance-of-the-spring-petclinic-sample-application-part-4-of- Last accessed March 8 2016.

Eclipse (2016a). Aspectj. http://eclipse.org/aspectj/. Last accessed March 16, 2016.

Eclipse (2016b). Eclipse Java development tools. https://eclipse.org/jdt/. Last accessed March 16, 2016.

Eclipse (2016c). Eclipselink. http://www.eclipse.org/eclipselink/. Last accessed March 16, 2016.

Eclipse (2016d). Eclipselink JPA 2.1. https://wiki.eclipse.org/EclipseLink/Release/2.5/JPA21#Entity_Graphs. Last accessed May 16, 2016.

EclipseLink (2016a). Eclipselink documentation. http://www.eclipse.org/eclipselink/documentation/2.5/solutions/migrhib002.htm. Last accessed March 16, 2016.

EclipseLink (2016b). Eclipselink documentation. http://eclipse.org/eclipselink/documentation/2.4/concepts/descriptors002.htm. Last accessed March 16, 2016.

Facebook (2016). Infer. http://fbinfer.com/. Last accessed March 8 2016.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. (2000). Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, **8**(3), 281–293.

Ferreira, V. (2016). Pitfalls of the Hibernate second-level / query caches. https://dzone.com/articles/pitfalls-hibernate-second-0. Last accessed March 3 2016.

Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux Journal*, **2004**(124), 1–5.

Fluri, B., Wursch, M., and Gall, H. C. (2007). Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, Vancouver, BC, Canada. IEEE Computer Society.

Fluri, B., Würsch, M., Giger, E., and Gall, H. C. (2009). Analyzing the co-evolution of comments and source code. *Software Quality Control*, **17**, 367–394.

Forum, H. U. (2004). Delete then insert in collection - order of executed sql. https://forum.hibernate.org/viewtopic.php?t=934483. Last accessed 15 Feb 2016.

FoundationDB (2015). FoundationDB. http://community.foundationdb.com/. Last accessed Feb 16, 2015.

Fu, Q., Lou, J.-G., Lin, Q., Ding, R., Zhang, D., Ye, Z., and Xie, T. (2012). Performance issue diagnosis for online service systems. In *Proceedings of the 31st International Symposium on Reliable Distributed Systems*, SRDS'12, pages 273–278.

Gall, H., Jazayeri, M., Klösch, R., and Trausmuth, G. (1997). Software Evolution Observations Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance*, pages 160–166.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76.

Gligoric, M. and Majumdar, R. (2013). Model checking database applications. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 549–564.

Gobert, M., Maes, J., Cleve, A., and Weber, J. (2013). Understanding schema evolution as a basis for database reengineering. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ICSM'13, pages 472–475.

Godfrey, M. W. and Tu, Q. (2000). Evolution in Open Source Software: A Case Study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, San Jose, California, USA. IEEE Computer Society.

Goeminne, M. and Mens, T. (2015). Towards a survival analysis of database framework usage in Java projects. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 551–555.

Goeminne, M., Decan, A., and Mens, T. (2014). Co-evolving code-related and database-related changes in a data-intensive software system. In *Proceedings of the Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week*, pages 353–357.

Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 395–404.

Gollmann, D. (2011). *Computer Security*. Wiley.

Google (2016). Error prone. http://errorprone.info/. Last accessed March 8 2016.

Gould, C., Su, Z., and Devanbu, P. (2004). Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 645–654.

Grechanik, M., Fu, C., and Xie, Q. (2012). Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166.

Grechanik, M., Hossain, B. M. M., Buy, U., and Wang, H. (2013a). Preventing database deadlocks in applications. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 356–366.

Grechanik, M., Hossain, B., and Buy, U. (2013b). Testing database-centric applications for causes of database deadlocks. In *Proceedings of the 6th International Conference on Software Testing Verification and Validation*, ICST '13, pages 174–183.

Grechanik, M., Luo, Q., Poshyvanyk, D., and Porter, A. (2016). Enhancing rules for cloud resource provisioning via learned software performance models. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 209–214.

Greevy, O., Ducasse, S., and Gîrba, T. (2006). Analyzing software evolution through feature views: Research Articles. *Journal of Software Maintenance and Evolution*, **18**, 425–456.

Grust, T., Mayr, M., Rittinger, J., and Schreiber, T. (2009). Ferry: Database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1063–1066.

Guravannavar, R. and Sudarshan, S. (2008). Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment*, **1**(1), 1107–1123.

Hartung, J., Knapp, G., and Sinha, B. (2011). *Statistical Meta-Analysis with Applications*. Wiley.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88.

Hassan, A. E. and Holt, R. C. (2006). Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, **11**(3), 335–367.

Henry, S. and Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, **SE-7**(5), 510–518.

Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. M. (2013). The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Survey*, **46**(2), 28:1–28:28.

His, I. and Potts, C. (2000). Studying the Evolution and Enhancement of Software Features. In *Proceedings of the International Conference on Software Maintenance*, pages 143–151.

Hoekstra, M. (2011). *Static source code analysis with respect to ORM performance antipatterns*. Master's thesis.

Hou, D. and Wang, Y. (2009). An empirical analysis of the evolution of user-visible features in an integrated development environment. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON'09, pages 122–135.

Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. *ACM SIGPLAN Notices*, **39**(12), 92–106.

IBM (2015). Restful web services: The basics. http://www.ibm.com/developerworks/library/ws-restful/. Last Accessed Mar 11 2016.

IBM (2016a). Security appscan source. http://www-03.ibm.com/software/products/en/appscan-source. Last accessed March 8 2016.

IBM (2016b). Websphere. http://www-01.ibm.com/software/ca/en/websphere/. Last accessed March 16, 2016.

Ibrahim, A. and Cook, W. R. (2006). Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 50–73.

Ibrahim, W. M., Bettenburg, N., Adams, B., and Hassan, A. E. (2012). On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, **85**(10), 2293–2304.

Iu, M.-Y. and Zwaenepoel, W. (2006). Queryll: Java database queries through bytecode rewriting. In *Middleware 2006*, volume 4290 of *Lecture Notes in Computer Science*, pages 201–218. Springer Berlin Heidelberg.

Iu, M.-Y., Cecchet, E., and Zwaenepoel, W. (2010). JReq: Database queries in imperative languages. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 84–103. Springer Berlin Heidelberg.

Iyer, S., Rowstron, A., and Druschel, P. (2002). Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 213–222.

JBoss (2016). Hibernate. http://www.hibernate.org/. Last accessed March 8 2016.

Jedyk, M. (2014). Transactions (mis)management: how to kill your app. http://www.resilientdatasystems.co.uk/java/transactions-mis-management-how-to-kill-app/. Last accessed 15 Feb 2016.

Jensen, K. (1997). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Coloured Petri Nets. Springer.

Jiang, Z. M. and Hassan, A. E. (2006). Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 179–180.

Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). Automatic identification of load testing problems. In *Proceedings of 24th IEEE International Conference on Software Maintenance*, ICSM'2008, pages 307–316.

Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. (2012). Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12.

Jin, S. and Bestavros, A. (2001). Greedydual* web caching algorithm: Exploiting

the two sources of temporal locality in web request streams. *Computer Communications*, **24**(2), 174–183.

Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681.

Johnson, R. (2005). J2EE development frameworks. *Computer*, **38**(1), 107–110.

Johnson, T. and Shasha, D. (1994). 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450.

Jovic, M., Adamoli, A., and Hauswirth, M. (2011). Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 155–170.

Jula, H., Tralamazza, D., Zamfir, C., and Candea, G. (2008). Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 295–308.

Kalibera, T. and Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 international symposium on International symposium on memory management*, ISMM '13, pages 63–74.

Kampenes, V. B., Dybå, T., Hannay, J. E., and Sjøberg, D. I. K. (2007). Systematic

review: A systematic review of effect size in software engineering experiments. *Information and Software Technology*, **49**(11-12), 1073–1086.

Kapfhammer, G. M., McMinn, P., and Wright, C. J. (2013). Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the 6th International Conference on Software Testing Verification and Validation*, ICST '13, pages 31–40.

Keith, M. and Stafford, R. (2008). Exposing the orm cache. *Queue*, **6**(3), 38–47.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, **28**(8), 721–734.

Kothari, J., Bespalov, D., Mancoridis, S., and Shokoufandeh, A. (2008). On evaluating the efficiency of software feature development using algebraic manifolds. In *Proceedings of the International Conference on Software Maintenance*, ICSM '08, pages 7–16.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, **1**(3), 26–49.

Leavitt, N. (2000). Whatever happened to object-oriented databases? *Computer*, **33**(8), 16–19.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and Laws of Software Evolution - The Nineties View. In *Proceedings of the 4th International Symposium on Software Metrics*, pages 20–32.

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, **32**(3), 176–192.

Linares-Vasquez, M., Li, B., Vendome, C., and Poshyvanyk, D. (2015). How do developers document database usages in source code? (N). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 36–41.

Linwood, J. and Minter, D. (2010). *Beginning Hibernate*. Apresspod Series. Apress.

Liu, X., Ma, Y., Liu, Y., Xie, T., and Huang, G. (2015). Demystifying the imperfect client-side cache performance of mobile web browsing. *IEEE Transactions on Mobile Computing*, **PP**(99).

Lo, D., Nagappan, N., and Zimmermann, T. (2015). How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 415–425.

Luo, Q., Nair, A., Grechanik, M., and Poshyvanyk, D. (2015). Forepost: finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, pages 1–51.

Malik, H., Chowdhury, I., Tsou, H.-M., Jiang, Z. M., and Hassan, A. E. (2008). Understanding the rationale for updating a function's comment. In *Proceedings of 24th IEEE International Conference on Software Maintenance*, ICSM '08, pages 167–176.

Manjhi, A., Garrod, C., Maggs, B. M., Mowry, T. C., and Tomasic, A. (2009). Holistic query transformations for dynamic web applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1175–1178.

McDonald, C. (2016). JPA performance, don't ignore the database. https://weblogs.java.net/blog/caroljmcdonald/archive/2009/08/28/jpa-performance-dont-ignore-database-0. Last accessed March 16, 2016.

Memcached (2016). Memcached. http://memcached.org/. Last accessed March 8 2016.

Meurice, L. and Cleve, A. (2014). Dahlia: A visual analyzer of database schema evolution. In *Proceedings of the Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 464–468.

MIHALCEA, V. (2014). A beginner's guide to JPA/Hibernate flush strategies. http://vladmihalcea.com/2014/08/07/a-beginners-guide-to-jpahibernate-flush-strategies/. Last accessed 15 Feb 2016.

Moore, D., MacCabe, G., and Craig, B. (2009). *Introduction to the Practice of Statistics*. W.H. Freeman and Company.

Nakagawa, S. and Cuthill, I. C. (2007). Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, **82**, 591–605.

Nanda, M. G., Gupta, M., Sinha, S., Chandra, S., Schmidt, D., and Balachandran, P. (2010). Making defect-finding tools work for you. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 99–108.

Nijjar, J. and Bultan, T. (2011). Bounded verification of Ruby on Rails data models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 67–77.

Nijjar, J. and Bultan, T. (2013). Data model property inference and repair. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 202–212.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling Memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 385–398.

Nistor, A., Song, L., Marinov, D., and Lu, S. (2013). Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571.

Nistor, A., Chang, P.-C., Radoi, C., and Lu, S. (2015). Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 2015 International Conference on Software Engineering*, ICSE '15, pages 902–912.

ObjectDB (2016a). JPA performance benchmark. http://www.jpab.org/All/All/All.html. Last accessed March 16, 2016.

ObjectDB (2016b). JPA2 annotations - the complete reference. http://www.objectdb.com/api/java/jpa/annotations. Last accessed March 10 2016.

OpenMRS (2016). OpenMRS. http://openmrs.org/. Last accessed March 8 2016.

Oracle (2015). Java ee platform specification. https://java.net/projects/javaee-spec/pages/Home. Last accessed Mar 11 2016.

Parsons, T. and Murphy, J. (2004). A framework for automatically detecting and assessing performance antipatterns in component based systems using run-time analysis. In *The 9th International Workshop on Component Oriented Programming*, WCOP '04, pages 1–7.

Paul, S. and Fei, Z. (2001). Distributed caching with centralized control. *Comput. Commun.*, **24**(2), 256–268.

PetClinic, S. (2016). Petclinic. https://github.com/SpringSource/spring-petclinic/. Last accessed March 8 2016.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR.

PMD (2016). Pmd. https://pmd.github.io/. Last accessed March 8 2016.

Pohjalainen, P. and Taina, J. (2008). Self-configuring object-to-relational mapping queries. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 53–59.

Portal, D. (2015). Devproof portal. https://code.google.com/p/devproof/. Last accessed June 1 2015.

Qiu, D., Li, B., and Su, Z. (2013). An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 125–135.

Rabkin, A. and Katz, R. (2011a). Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 193–202.

Rabkin, A. and Katz, R. (2011b). Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 131–140.

Rails (2016). What's new in edge rails partial updates. http://archives.ryandaigle.com/articles/2008/4/1/what-s-new-in-edge-rails-partial-updates. Last accessed March 16, 2016.

Ramachandra, K. and Sudarshan, S. (2012). Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 133–144.

Ramachandra, K., Guravannavar, R., and Sudarshan, S. (2012). Program analysis and transformation for holistic optimization of database applications. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 39–44.

Ramachandra, K., Chavan, M., Guravannavar, R., and Sudarshan, S. (2015). Program transformations for asynchronous and batched query submission. *IEEE Transactions on Knowledge and Data Engineering*, **27**(2), 531–544.

Richardson, L. and Ruby, S. (2008). *RESTful Web Services*. O'Reilly Media.

Rohr, M., van Hoorn, A., Matevska, J., Sommer, N., Stoever, L., Giesecke, S., and Hasselbring, W. (2008). Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 80–85.

Romano, J., Kromrey, J., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, pages 1–3.

Shang, W., Jiang, Z. M., Adams, B., Hassan, A. E., Godfrey, M. W., Nasser, M., and Flora, P. (2011). An exploratory study of the evolution of communicated information about the execution of large software systems. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 335–344.

Shang, W., Jiang, Z. M., Hemmati, H., Adams, B., Hassan, A. E., and Martin, P. (2013). Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411.

Shang, W., Jiang, Z. M., Adams, B., Hassan, A. E., Godfrey, M. W., Nasser, M., and Flora, P. (2014). An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, **26**(1), 3–26.

Shen, D., Luo, Q., Poshyvanyk, D., and Grechanik, M. (2015). Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA

2015, pages 270–281.

Shen, H., Fang, J., and Zhao, J. (2011). EFindBugs: Effective error ranking for FindBugs. In *Proceedings of the 2011 IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 299–308.

Smith, A. J. (1985). Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 64–73.

Smith, C. and Williams, L. (2001). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. The Addison-Wesley object technology series. ADDISON WESLEY Publishing Company Incorporated.

Smith, C. U. and Williams, L. (2003). More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Proceedings of the 2003 Computer Measurement Group Conference*, CMG 2003, pages 1–9.

Smith, C. U. and Williams, L. G. (2000). Software performance antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136.

Smith, J., Johnson, B., Murphy-Hill, E., Chu, B., and Lipford, H. R. (2015). Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259.

Soh, Z., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2013). Towards understanding how developers spend their effort during maintenance activities. In

Proceedings of the 2013 Working Conference on Reverse Engineering, WCRE '13, pages 152–161.

SpringSource (2016). Spring framework. www.springsource.org/. Last accessed 25 July 2016.

StackOverflow (2009). Spring @Transactional read-only propagation. http://stackoverflow.com/questions/1614139/spring-transactional-read-only-propagation. Last accessed 15 Feb 2016.

StackOverflow (2010a). Spring transaction readonly. http://stackoverflow.com/questions/2562865/spring-transaction-readonly. Last accessed 15 Feb 2016.

StackOverflow (2010b). Starting new transaction in spring bean. http://stackoverflow.com/questions/3037006/starting-new-transaction-in-spring-bean. Last accessed 15 Feb 2016.

StackOverflow (2014). Spring transaction: requires_new beharivour. http://stackoverflow.com/questions/22927763/spring-transaction-requires-new-beharivour. Last accessed 15 Feb 2016.

StackOverflow (2015). How to change the ordering of SQL execution in Hibernate. http://stackoverflow.com/questions/20395543/how-to-change-the-ordering-of-sql-execution-in-\hibernate. Last accessed 15 Feb 2016.

StackOverflow (2016a). Django objects values select only some fields. http://stackoverflow.com/questions/7071352/django-objects-values-select-only-some-fields. Last accessed March 16, 2016.

StackOverflow (2016b). Hibernate : dynamic-update dynamic-insert - performance effects. http://stackoverflow.com/questions/3404630/hibernate-dynamic-update-dynamic-insert\-performance-effects?lq=1. Last accessed March 16, 2016.

StackOverflow (2016c). Hibernate criteria query to get specific columns. http://stackoverflow.com/questions/11626761/hibernate-criteria-query-to-get-specific-columns. Last accessed March 16, 2016.

StackOverflow (2016d). JPA2.0/hibernate: Why JPA fires query to update all columns value even some states of managed beans are changed? http://stackoverflow.com/questions/15760934/jpa2-0-hibernate-why-jpa-fires-query-to-update\-all-columns-value-even-some-stat. Last accessed March 16, 2016.

StackOverflow (2016e). Making a OneToOne-relation lazy. http://stackoverflow.com/questions/1444227/making-a-onetoone-relation-lazy. Last accessed March 16, 2016.

StackoverFlow (2016). MySQL - how many columns is too many? http://stackoverflow.com/questions/3184478/how-many-columns-is-too-many-columns. Last accessed March 16, 2016.

Sutherland, J. and Clarke, D. (2016). Java persistence. https://en.wikibooks.org/wiki/Java_Persistence. Last accessed March 16, 2016.

Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M., and Flora, P. (2014). Continuous validation of load test suites. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 259–270.

Tamayo, J. M., Aiken, A., Bronson, N., and Sagiv, M. (2012). Understanding the behavior of database operations under program control. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 983–996.

Terracotta (2016). Ehcache. http://ehcache.org/. Last accessed March 8 2016.

ThinkGem (2016). JEEsite. http://jeesite.com/. Last accessed June 13 2016.

Tianyin, Jin, L., Fan, X., and Zhou, Y. (2015). Hey, you have given me too many knobs! understanding and dealing with over-designed configuration in system software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '15.

Tomcat, A. (2016). Logging in tomcat. https://tomcat.apache.org/tomcat-8.0-doc/logging.html. Last accessed March 8 2016.

TPCW (2016). Transactional web e-commerce benchmark. http://www.tpc.org/tpcw/. Last accessed March 3 2016.

van Hoorn, A., Rohr, M., and Hasselbring, W. (2008). Generating probabilistic and intensity-varying workload for web-based software systems. In *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 124–143.

Wegrzynowicz, P. (2013). Performance anti-patterns in hibernate. http://www.devoxx.com/display/DV11/Performance+Anti-Patterns+in+Hibernate. Last accessed April 10 2013.

Wiedermann, B. and Cook, W. R. (2007). Extracting queries by static analysis of transparent persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 199–210.

Wiedermann, B., Ibrahim, A., and Cook, W. R. (2008). Interprocedural query extraction for transparent persistence. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 19–36.

Xiao, X., Han, S., Zhang, D., and Xie, T. (2013). Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 90–100.

Xiong, Y., Hubaux, A., She, S., and Czarnecki, K. (2012). Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 58–68.

Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., and Sevitsky, G.

(2010a). Finding low-utility data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 174–186.

Xu, G., Mitchell, N., Arnold, M., Rountev, A., and Sevitsky, G. (2010b). Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 421–426.

Yagoub, K., Belknap, P., Dageville, B., Dias, K., Joshi, S., and Yu, H. (2008). Oracle's SQL Performance Analyzer. *IEEE Data Engineering Bulletin*.

Yan, C., Chu, Z., Cheung, A., and Lu, S. (2016). Database-backed applications in the wild: How well do they work? *arXiv preprint arXiv:1607.02561*.

Zaitsev, P., Tkachenko, V., Zawodny, J., Lentz, A., and Balling, D. (2008). *High Performance MySQL: Optimization, Backups, Replication, and More*. O'Reilly Media.

Zaman, S., Adams, B., and Hassan, A. E. (2011). Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102.

Zaparanuks, D. and Hauswirth, M. (2012). Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76.

ZeroturnAround (2014). Java tools and technologies landscape for 2015. http://zeroturnaround.com/rebellabs/

java-tools-and-technologies-landscape-for-2014/. Last accessed March 8 2016.

Zhang, H., Tan, H. B. K., Zhang, L., Lin, X., Wang, X., Zhang, C., and Mei, H. (2011). Checking enforcement of integrity constraints in database applications based on code patterns. *Journal of Systems and Software*, **84**(12), 2253 – 2264.

Zhang, S. and Ernst, M. D. (2013). Automated diagnosis of software configuration errors. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 312–321.

Zhang, S. and Ernst, M. D. (2014). Which configuration option should I change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 152–163.

Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., and Stumm, M. (2014). Lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644. USENIX Association.

Zhou, Y., Philbin, J., and Li, K. (2001). The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 91–104.