Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks

Tse-Hsun Chen, *Student Member, IEEE,* Weiyi Shang, *Member, IEEE,* Zhen Ming Jiang, *Member, IEEE,* Ahmed E. Hassan, *Member, IEEE,* Mohamed Nasser, *Member, IEEE,* and Parminder Flora, *Member, IEEE,*

Abstract—Developers usually leverage Object-Relational Mapping (ORM) to abstract complex database accesses for large-scale systems. However, since ORM frameworks operate at a lower-level (i.e., data access), ORM frameworks do not know how the data will be used when returned from database management systems (DBMSs). Therefore, ORM cannot provide an optimal data retrieval approach for all applications, which may result in accessing redundant data and significantly affect system performance. Although ORM frameworks provide ways to resolve redundant data problems, due to the complexity of modern systems, developers may not be able to locate such problems in the code; hence, may not proactively resolve the problems. In this paper, we propose an automated approach, which we implement as a Java framework, to locate redundant data problems. We apply our framework on one enterprise and two open source systems. We find that redundant data problems exist in 87% of the exercised transactions. Due to the large number of detected redundant data problems, 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 problems, the system response time for the studied systems can be improved by an average of 17%.

Index Terms—Performance, ORM, Object-Relational Mapping, Program analysis, Database.

1 Introduction

Due to the increasing popularity of big data applications and cloud computing, software systems are becoming more dependent on the underlying database for data management and analysis. As a system becomes more complex, developers start to leverage technologies to manage the data consistency between the source code and the database management systems (DBMSs).

One of the most popular technologies that developers use to help them manage data is Object-Relational Mapping (ORM) framework. ORM frameworks provide a conceptual abstraction for mapping database records to objects in object-oriented languages [43]. With ORM, objects are directly mapped to database records. For example, to update a user's name in the database, a simple method call user.updateName("Peter") is needed. By adopting ORM technology, developers can focus on the high-level business logic without worrying about the underlying database access details and without having to write error-prone database boilerplate code [10], [47].

ORM has become very popular among developers since early 2000, and its popularity continues to rise in practice [39]. For instance, there exists ORM frameworks for most modern Object-Oriented programming languages such as Java, C#, and Python. However, despite ORM's advantages and popularity, there exist redundant data problems in ORM frameworks [4], [5], [8], [24]. Such redundant data problems are usually caused by non-optimal use of ORM frameworks.

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 systems 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:

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

- *T-H. Chen, A. E. Hassan are with the Software Analysis and Intelligence Lab (SAIL) in the School of Computing at Queen's University, Canada. E-mail: {tsehsun, ahmed}@cs.queensu.ca*
- *W. Shang is with Concordia University, Canada. E-mail: shang@encs.concordia.ca*
- *Z. Jiang is with York University, Canada. E-mail: zmjiang@cse.yorku.ca*
- *M. Nasser and P. Flora are with BlackBerry, Canada.*

Manuscript received April 19, 2005; revised September 17, 2014.

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 problems 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 [69]. The redundant data problems 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 [50], [58] have shown that the number of columns in a table can be very large in real-world systems (e.g., the tables in the OSCAR database have 30 columns *on average* [50]), and some systems may even have tables with more than 500 columns [7]. Thus, locating redundant data problems is helpful for large-scale real-world systems.

In fact, developers have shown that by optimizing ORM configurations and data retrieval, system performance can increase by as much as 10 folds [15], [63]. However, even though developers can change ORM code configurations to resolve different kinds of redundant data problems, due to the complexity of software systems, developers may not be able to locate such problems in the code, and thus may not proactively resolve the problems [15], [40]. Besides, there is no guarantee that every developer knows the impact of such problems.

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

Redundant data or computation is a well-known cause for performance problems [53], [54], and in this paper, we focus on detecting database-related redundant data problems. Our approach consists of both static and dynamic analysis. We first apply static analysis on the source code to automatically identify database-accessing functions (i.e., functions that may access the data in the DBMS). Then, we use bytecode instrumentation on the system executables to obtain the code execution traces and the ORM generated SQL queries. We identify the needed database accesses by finding which database-accessing functions are called during the system execution. We identify the requested database accesses by analyzing the ORM generated SQL queries. Finally, we discover instances of the redundant data problems 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 systems (Pet Clinic [57] and Broadleaf Commerce [21]) and one large-scale Enterprise System (ES). We find that redundant data problems exist in all of our exercised workloads. In addition, our statistical rigorous performance assessment [33] shows that resolving redundant data problems can improve the system performance (i.e., response time) of the studied systems by 2–92%, depending on the workload. Our performance assessment approach can further help developers prioritize the efforts for resolving the redundant data problems according to their performance impact.

The main contributions of this paper are:

1) We survey the redundant data problems in popular ORM frameworks across four different programming languages, and we find that the different popular frameworks share common problems.
2) We propose an automated approach to locate the redundant data problems in ORM frameworks, and we have implemented a Java-version to detect redundant data problems in Java systems.
3) Case studies on two open source and one enterprise system (ES) show that resolving redundant data problems can improve the system 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 ES developers, and is now integrated into the performance testing process for the ES.

Paper Organization. The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 discusses the background knowledge of ORM. Section 4 describes our approach for finding redundant data problems. Section 5 provides the background of our case study systems, and the experimental setup. Section 6 discusses our framework implementation, the types and the prevalence of redundant data problems that we discovered, and introduces our performance assessment approach and the results of our case studies. Section 7 surveys the studied redundant data problems in different ORM frameworks. Section 8 talks about the threats to validity. Finally, Section 9 concludes the paper.

2 Related Work

In this section, we discuss related prior research.

Optimizing DBMS-based Applications. Many prior studies aim to improve system performance by optimizing how systems access or communicate with a DBMS. Cheung *et al*. [17] propose an approach to delay all queries as late as possible so that more queries can be sent to the DBMS in a batch. Ramachandra *et al*. [59], on the other hand, prefetch all the data at the beginning to improve system performance. Chavan *et al*. [14] automatically transform query execution code so that queries can be sent to the DBMS in an asynchronous fashion. Therefore, the performance impact of data and query transmission can be minimized. Bowman *et al*. [12] optimize system performance by predicting repeated SQL patterns. They develop a system on top of DBMS client

libraries, and their system can automatically learn the SQL patterns, and transform the SQLs into a more optimized form (e.g., combine loop-generated SQL selects into one SQL).

Our paper's goal is to improve system performance by finding redundant data problems in systems that are developed using ORM frameworks. Our approach can reduce unnecessary data transmission and DBMS computation. Different from prior work, our approach does not introduce another layer to existing systems, which increases system complexity, but rather our approach pinpoints the problems to developers. Developers can then decide a series of actions to prioritize and resolve the redundant data problems.

Detecting Performance Bugs. Prior studies propose various approaches to detect different performance bugs through run-time indicators of such bugs. Nistor *et al*. [54] propose a performance bug detection tool, which detects performance problems by finding similar memory-access patterns during system execution. Chis *et al*. [19] provide a tool to detect memory anti-patterns in Java heap dumps using a catalogue. Parsons *et al*. [56] present an approach for automatically detecting performance issues in enterprise applications that are developed using component-based frameworks. Parsons *et al*. detect performance issues by reconstructing the run-time design of the system using monitoring and analysis approaches.

Xu *et al*. [68] 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*. [66] use different workflows to identify and predict workflow-dependent performance bottlenecks (i.e., performance bugs) in GUI applications. Xu *et al*. [67] introduce a run-time analysis to identify low-utility data structures whose costs are out of line with their gained benefits. Grechanik *et al*. develop various approaches for detecting and preventing database deadlocks through static and dynamic analysis [35], [36]. Chaudhuri *et al*. [13] propose an approach to map the DBMS profiler and the code for finding the root causes of slow database operations. Similar to prior studies, our approach relies on dynamic system information. However, we focus on systems that use ORM frameworks to map code to DBMSs.

In our prior research, we propose a framework to statically identify performance anti-patterns by analyzing the system source code [15]. This paper is different from our prior study in many aspects. First, in our prior study, we develop a framework for detecting two performance anti-patterns that we observed in practice. Only one of these performance anti-patterns is related to data retrieval. In this paper, we focus on the redundant data problems between the needed data in the code and the SQL requested data. Performance anti-patterns and redundant data problems are two different sets of problems with little overlap. Performance anti-patterns may be any code patterns that may result in bad performance. The problem can be related to memory, CPU, network, or database. On the other hand, redundant data problems are usually caused by requesting/updating too much data than actually needed.

We propose an approach to locate such redundant data problems, and we do not know what kinds of redundant data problems are there before applying our approach. Second, in our prior study, we use only static analysis for detecting performance anti-patterns. However, static analysis is prone to false positives as it is difficult to obtain an accurate data flow and pointer analysis given the assumptions made during computation [16]. Thus, most of the problems we study in this paper cannot be detected by simply extending our prior framework. In this paper, we propose a hybrid approach using both static and dynamic analysis to locate the redundant data problems in the code. Our hybrid approach can give more precise results and better locate the problems in the code. In addition, we implemented a tool to transform SQL queries into abstract syntax trees for further analysis. Finally, we manually classify and document the redundant data problems that we discovered, and we conduct a survey on their existence in ORM frameworks across different programming languages.

3 Background

In this section, we provide some background knowledge of ORM before introducing our approach. We first provide a brief overview of different ORM frameworks, and then we discuss how ORM accesses the DBMS using an example. 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 of ORM

ORM has become very popular among developers due to its convenience [39], [47]. Most modern programming languages, such as Java, C#, Ruby, and Python, all support ORM. 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 systems [64]. Using JPA, users can switch between different ORM providers with minimal modifications.

There are many implementations of JPA, such as Hibernate [22], OpenJPA [28], EclipseLink [32], and parts of IBM WebSphere [37]. 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 [48]). In this paper, we implement our approach as a framework for detecting redundant data problems for JPA systems due to the popularity of JPA.

3.2 Translating Objects to SQL Queries

ORM is responsible for mapping and translating database entity objects to/from database records. Figure 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 source code annotations. There are three categories of source code annotations:

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

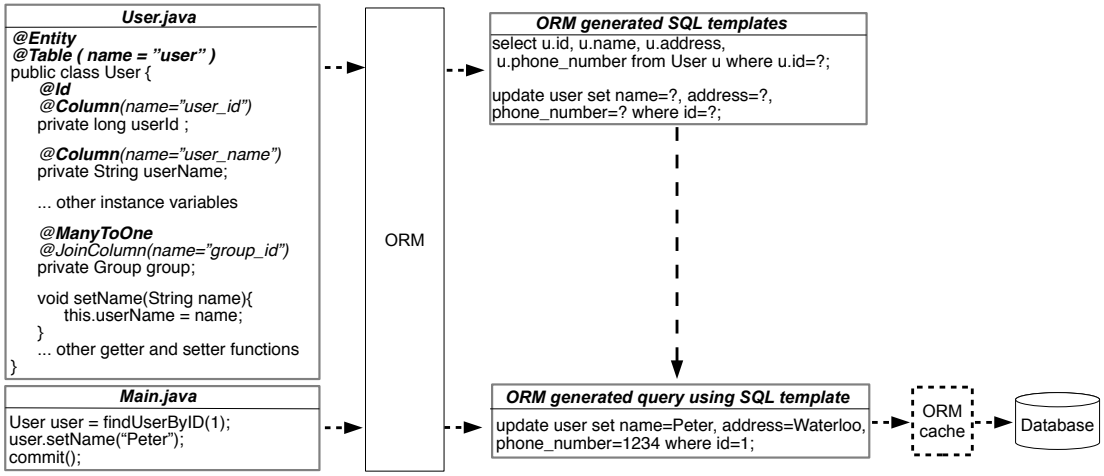

Fig. 1. An example flow of how JPA translates object manipulation to SQL. Although the syntax and configurations may be different for other ORM frameworks, the fundamental idea is the same: developers need to specify the mapping between objects and database tables, the relationships between objects, and the data retrieval configuration (e.g., eager v.s. lazy).

to a database table (marked as **@Table** in the source 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 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 [30].

JPA generates and may cache SQL templates (depending on the implementation) for each database entity class. 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 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.

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 [44]. 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 cache mechanism provides significant performance improvement to systems that rely heavily on DBMSs.

Our case study systems use Hibernate [22] as the JPA implementation due to Hibernate's popularity. However, as shown in Section 7, our survey finds that redundant data problems also exist in other ORM frameworks and are not specific to the JPA implementation that we choose.

4 Our Approach of Finding Redundant Data Problems

In the previous section, we introduce how ORM frameworks map objects to database records. However, such mapping is 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 locating the redundant data problem 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).

4.1 Overview of Our Approach

Figure 2 shows an overview of our approach for locating redundant data problems. We define the *needed database accesses* as how database-accessing functions are called during system execution. We define the *requested database accesses* as the corresponding generated SQL queries during system execution. Our approach consists of three different phases. First, we use static source code analysis to automatically identify the database-accessing functions (functions that read or modify instance variables that are mapped to database columns). Second, we leverage bytecode instrumentation to monitor and collect system execution traces. In particular, we collect the exercised database-accessing

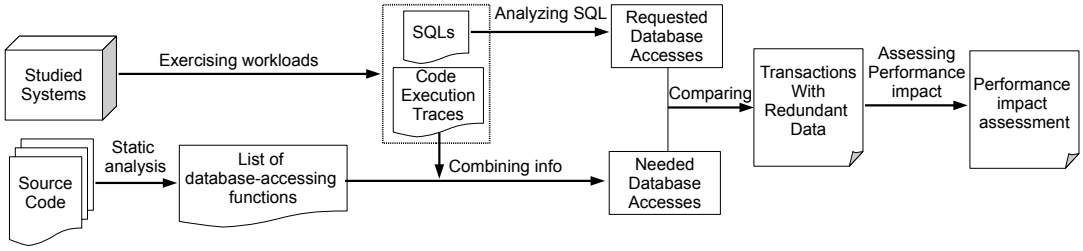

Fig. 2. An overview of our approach for finding and evaluating redundant data problems.

functions (and the location of the call site of such functions) as well as the generated SQLs. Finally, we find the redundant data problems by comparing the exercised database-accessing functions and the SQLs. We explain the detail of each phase in the following subsections.

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 functions along a function call graph that may read or modify a given variable. If a database instance variable is modified in a function, we consider the function as a *data-write function*. If a database instance variable is being read or returned in a function, we consider the function as a *data-read function*. 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 function getUserName(), which returns the variable *name*, is a data-read function. 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.* [23]. 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 data-accessing function from its superclass, we use the result of the class inheritance graph to determine the columns that the subclass function is accessing.

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 [2], 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.

```
<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>
```

Fig. 3. An example of the exercised database-accessing functions and generated SQL queries during a transaction.

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 3 shows an example of the resulting data. In that XML snippet, the function call user.getUserName() (the needed data access) is translated to a select SQL (the requested data access) in a transaction.

We find redundant data problems 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 function is accessing, we compare the column reads and writes between the SQL queries and the database-accessing functions. If a column that is being selected/updated in an SQL query has no corresponding function that reads/updates the column, then the transaction has a redundant data problem (e.g., in Figure 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.

4.5 Performance Assessment

We propose an approach to automatically assess the performance impact of the redundant data problems. The performance assessment results can be used to prioritize

TABLE 1
Statistics of the studied systems.

System	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
ES	> 300K	> 3,000	> 50

performance optimization efforts. Since there may be different types of redundant data problems and each type may need to be assessed differently, we discuss our assessment approach in detail in Section 6.3, after discussing the types of redundant data problems that we discovered in Section 6.2.

5 Experimental Setup

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

5.1 Case Study Systems

In this paper, we implement our approach as a framework, and apply the framework on two open-source systems (Pet Clinic [57] and Broadleaf Commerce [21]) and one large-scale Enterprise System (ES). Pet Clinic is a system developed by Spring [62], 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 [15], [34], [38], [60], [65]. Broadleaf [21] is a large open source e-commerce system that is widely used in both non-commercial and commercial settings worldwide. ES 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 ES due to a non-disclosure agreement (NDA), we also conduct our study on two open source systems. Table 1 shows the statistics of the three studied systems.

All of our studied systems are web systems 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 [18]). The studied systems follow the typical "Model-View-Controller" design pattern [46], and use Spring [62] to manage HTTP requests. We use MySQL as the DBMS in our experiment.

5.2 Experiments

Our approach and framework require dynamic analysis. However, since it is difficult to generate representative workloads (i.e., system use cases) for a system, we use the readily available performance test suites in the studied systems (i.e., Pet Clinic and ES) to obtain the execution traces. If the performance test suites are not present (i.e., Broadleaf), we use the integration test suites as an alternate choice. Both the performance and the integration test suites are designed to test different *features* in a system (i.e., use case testing). Both performance and integration test suites provide more realistic workloads and better test coverage [11]. Table 3 shows the descriptions of the exercised test suites. Nevertheless, our approach can be adapted to deployed systems or to monitor real-world workloads for finding redundant data problems in production.

TABLE 2
Overview of the redundant data problems that we discovered in our exercised workloads. *Trans.* column shows where the redundant data problem 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)

We group the test execution traces according to the transactions to which they belong. Typically in database-related systems, 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).

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 problems that are discovered by our framework, and their performance assessment. We want to know if our approach can discover redundant data problems. If so, we want to also study what are the common redundant data problems and their prevalence in the studied systems. Finally, we assess the performance impact of the discovered redundant data problems.

6.1 Framework Implementation

To evaluate our approach, we implement our approach as a Java framework to detect redundant data problems in three studied JPA systems. We implement our static analysis tool for finding the needed database accesses using JDT [29]. We use AspectJ [31] to perform bytecode instrumentation on the studied systems. We instrument all the database-accessing functions 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 3).

6.2 Case Study Results

Using our framework, we are able to find a large number of redundant data problems in the studied systems. In fact, on average 87% of the exercised transactions contain at least one redundant data problem. Our approach is able to find the redundant data problems in the code, but we are also interested in understanding what kinds of redundant data problems are there. Moreover, we use the discovered redundant data problems to illustrate the performance impact of the redundant data problems. However, other types of redundant data problems may still be discovered using our approach, and the types of the redundant data problems

TABLE 3
Prevalence of the discovered redundant data problems in each test suite. The detail of ES is not shown due to NDA.

System	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%)
ES	Multiple Features	> 1000	$> 30\%$	3%	100%	0%	23%

that we study here is by no means complete. In the following subsections, we first describe the type of redundant data problems that we discovered, then we discuss their prevalence in our studied systems.

6.2.1 Types of Redundant Data Problems

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% [51]) in the exercised test suites that contain at least one redundant data problem (as shown in Table 3). We find that most redundant data can be grouped into four types, which we call: *update all*, *select all*, *excessive data*, and *per-transaction cache* (other types of redundant data problems may still exist, and may be discovered using our approach). Table 2 shows an overview of the redundant data problems 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 1). The redundant data problem is between the translations from objects to SQLs, where ORM simply updates all the database columns. This redundant data problem 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 problem [5], [9]. 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 problem can cause significant performance impact when the generated SQLs are updating unmodified non-cluster indexed columns [69]. Prior studies [50], [58] have shown that the number of columns in a table can be very large in real-world systems (e.g., the tables in the OSCAR database have *on average* 30 columns [50]), and some systems may even have tables with more than 500 columns [7]. Even in our studied systems, we find that some tables have more than 28, or even 50 columns (Table 1). Thus, this type of redundant data problem may be more problematic in large-scale systems.

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 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 performance overheads.

Developers also discuss the performance impact of this type of redundant data problem [4], [49]. For example, developers are complaining that the size of some columns is too large, and retrieving them from the database causes performance issues [4]. 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 problem 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* problem can improve system performance significantly [25]. Different ORM frameworks provide different ways to resolve this redundant data problem, and our approach can provide guidance for developers on this type of problem.

Per-Transaction Cache. Our approach described in Section 4 also looks for redundant data problems 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 a prior study [15]. *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), but the cache configuration is never automatically optimized for different application systems [44]. 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 [64], but they may not proactively leverage the benefit of such a cache. We have seen cases in real-world large-scale systems where this redundant data problem 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 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 can assist developers with cache optimization.

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

TABLE 4
Total number of SQLs and the number of duplicated selects in each test suite.

System	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%)
ES	Multiple Features	$>>$ 10,000	$>$ 10%

6.2.2 Prevalence of Redundant Data Problems

Table 3 shows the prevalence of the redundant data problems in the executed test suites (a transaction may have more than one redundant data problem). Due to NDA, we cannot show the detail results for ES. However, we see that many transactions ($>$30%) have an instance of a redundant data problem in ES.

Most exercised transactions in BroadLeaf and Pet Clinic have at least one instance of redundant data problem (e.g., at least 75% of the transactions in the five test suites 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 suites 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* problem 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 suite (Table 4). We filter out the duplicated selects where the selected data is modified. We find that some test suites have a larger number of duplicate selects than others. In Pet Clinic, we find that the *per-transaction cache* problems 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* problem.

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

6.3 Automated Performance Assessment

Since every ORM framework has different ways to resolve the redundant data problems, it is impossible to provide an automated ORM optimization for all systems. Yet, ORM

optimization requires a great amount of effort and a deep understanding of the system workloads and design. Thus, to reduce developers' effort on resolving the redundant data problems, we propose a performance assessment approach to help developers prioritize their performance optimization efforts.

6.3.1 Assessing the Performance Impact of Redundant Data Problems

We follow a similar methodology as a previous study [40] to automatically assess the performance impact of the redundant data problem. Note that our assessment approach is only for estimating the performance impact of redundant data problems in different workloads, and cannot completely fix the problems. Developers may wish to resolve these problems after further investigation. Below, we discuss the approaches that we use to assess each type of the discovered redundant data problems.

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 suites. 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 problem.

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 suites. 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*.

6.3.2 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 system execution, and such variances may lead to incorrect results [33], [41]. As a result, it is important to provide confidence intervals for performance measurements. We follow the recommendation of Georges *et al.* [33] and repeat each performance test for 30 times. We use a *Student's t-test* to determine if resolving a redundant data problem can result in a statistically significant (p-value $\leq$ 0.05) performance improvement. Although the *t-test* requires the population to be normally distributed, according to the Central Limit Theorem, our performance measurements will be approximately normal (we repeat the same test under the same environment for 30 times) [33], [51].

In addition, we calculate the *effect sizes* [42], [52] of the response time differences. Unlike the *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. Reporting only the statistical significance may lead to erroneous results [42] (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 [42]. *Cohen's d* measures the effect size statistically, and has been used in prior engineering studies [42], [45]. The strength of the effects and the corresponding range of *Cohen's d* values are [20]:

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

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 5 shows the assessed performance improvement after resolving each type of redundant data problem in each performance test suite. For each test suite, we report the total response time (in seconds) along with a confidence interval. In almost all test suites, resolving the *update all* problem 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 suite is related to browsing, which only performs a very small number of updates (only six update SQL queries). ES also does not have a significant improvement after resolving *update all*.

As discussed in Section 6.2, the *update all* problem 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 [61]. 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.

TABLE 5
Performance impact study by resolving the redundant data problems in each test suite. Response time is measured in seconds at the client side. We mark the results in bold if resolving the redundant data problems has a statistically significant improvement. For response time differences, large/medium/small/trivial effect sizes are marked with L, M, S, and T, respectively.

System	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**
ES	Multiple Features	—	0%	>0.05	**> 30%^{L}**	**<<0.001**	—	—	**> 30%^{L}**	**<<0.001**

Select All. The *select all* problem causes a statistically significant performance impact in Pet Clinic, ES, and two test suites in Broadleaf (3–31% improvement) with varying effect sizes. Due to the nature of the Broadleaf test suites, some columns have null values, which reduce the overhead of data transmission. Thus, the effect of the *select all* problem is not as significant as the *update all* problem. In addition to what we discuss in Section 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* problem has a high performance impact in Pet Clinic (92% performance improvement), but only 2–6% improvement in Broadleaf and 5% in ES with mostly non-trivial effect sizes. Since we know that the performance impact of the redundant data problem 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 [6]. 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* problem has a statistically significant performance impact in 4 out of 5 test suites in Broadleaf with non-trivial effect sizes. We also see a large performance improvement in Pet Clinic, where resolving the *per-transaction cache* problem improves the performance by 88%. Resolving the *per-transaction cache* problem also improves the ES 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 [69]. 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.

TABLE 6
Existence of the studied redundant data problems 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

All of our uncovered redundant data problems have a performance impact in all studied systems. Depending on the workloads, resolving the redundant data problems can improve the performance by up to 92% (17% on average). Our approach can automatically flag redundant data problems that have a statistically significant performance impact, and developers may use our approach to prioritize their performance optimization efforts.

7 A Survey on the Redundant Data Problems in Other ORM Frameworks

In previous sections, we apply our approach on the studied systems. We discover four types of redundant data problems, and we further illustrate their performance impact. However, since we only evaluate our approach on the studied systems, we do not know if the discovered redundant data problems 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 problems.

We study the documents on the ORM frameworks' official websites, and search for developer discussions about the redundant data problems. Table 6 shows the existence of the studied redundant data problems in the surveyed ORM frameworks under default configurations. Our studied systems use Hibernate 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* problem, but the problem does not exist in EclipseLink and ActiveRecord [26], [55]. These two ORM frameworks keep track of which columns are modified and only update the modified columns. This is the design tradeoff that the ORM developers made. The pros of the design decision is that this redundant data problem is handled by default. However, this will also introduce overheads such as tracking modifications and generating different SQLs for each update [3]. 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* problem. The reason may be the ORM implementation difficulties, since ORM frameworks do not know how the retrieved data will be used in the system. Nevertheless, all the surveyed ORM frameworks provide some way to retrieve only the needed data from the DBMS (e.g., [1], [27]).

Excessive data. All of the surveyed ORM frameworks may have the *excessive data* problem. However, some ORM frameworks handle this problem 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 [64].

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 systems, it is difficult to find a balance point between performance and stale data when using caches. Solving the problem 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 system performance.

8 Threats to Validity

In this section, we discuss the potential threats to validity of our work.

8.1 External Validity

We conduct our case study on three systems. Some of the redundant data problems may not exist in other systems, and we might also miss some problems. We try to address this problem by picking systems with various sizes, and include both open source and industrial systems in our study. Nevertheless, the most important part of our approach is that it can be adapted to find redundant data problems in other systems using various ORM frameworks.

8.2 Construct Validity

Manual Classification of the Redundant Data Problems. Our case study includes manual classification of the types of redundant data between the source code and the generated SQLs. We evaluate our discovered types of redundant data problems by studying their prevalence in the exercised workflows, but our findings may contain subjective bias and we may miss other types of redundant data problems. Nevertheless, we study the redundant data problems that are discovered in our studied systems to illustrate the impact of redundant data problems, but our approach is applicable to other systems.

Experimental Setup. We use either the performance or the integration test suites to exercise different workflows of the studied systems. These test suites may not cover all the workflows in the systems, and the workflows may not all be representative to real system workflows. However, our approach can be easily adapted to other workflows. The studied workflows demonstrate the feasibility and usefulness of our approach.

The redundant data problems that are studied in this paper may have different performance impact in other workflows, yet we have shown that developers indeed care about the impact of these redundant data problems. We conduct performance assessments in an experimental environment, which may also lead to different results compared to the performance impact in real-world environment. However, the improvements should be very similar in the production environment given that other conditions such as hardware are the same. Moreover, resolving the redundant data problem is challenging, as it requires a deep understanding of the system design and workflows. For example, one needs to first locate the workflows that have redundant data problems, and customize ORM configurations for the workflows based on the logic in the source code. Our proposed approach can provide an initial performance assessment, and the results can be used to assist developers in prioritizing their performance optimization efforts.

Redundant Data Problems in Different ORM Frameworks. Section 7 provides a survey on the redundant data problems in different ORM frameworks. We do not include the fixes of the redundant data problems, but the fixes are very similar across ORM frameworks. Moreover, the fixes are available in the frameworks' official documents. Although we did not survey the impact of the redundant data problems, the impact should be similar across ORM frameworks.

9 Conclusion

Object-Relational Mapping (ORM) frameworks provide a conceptual abstraction for mapping the source code to the DBMS. Since ORM automatically translates object accesses and manipulations to database queries, ORM significantly

simplifies software development. Thus, developers can focus on business logic instead of worrying about non-trivial database access details. However, ORM mappings introduce redundant data problems (e.g., the needed data in the code does not match with the requested data by the ORM framework), which may cause serious performance problems.

In this paper, we proposed an automated approach to locate the redundant data problems in the code. We also proposed an automated approach for helping developers prioritize the efforts on fixing the redundant data problems. We conducted a case study on two open source and one enterprise system to evaluate our approaches. We found that, depending on the workflow, all the redundant data problems that are discussed in the paper have statistically significant performance overheads, and developers are concerned about the impacts of these redundant data problems. Developers do not need to manually locate the redundant data problems in thousands of lines of code, and can leverage our approach to automatically locate and prioritize the effort to fix these redundant data problems.

ACKNOWLEDGMENTS

We are grateful to BlackBerry for providing access to the enterprise system used in our case study. The finding and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliation. Our results do not in any way reflect the quality of BlackBerry's products.

REFERENCES

[1] Django objects values select only some fields. http://stackoverflow.com/questions/7071352/django-objects-values-select-only-some-fields. Last accessed March 16, 2016.
[2] FoundationDB. http://community.foundationdb.com/. Last accessed May 16, 2016.
[3] 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.
[4] Hibernate criteria query to get specific columns. http://stackoverflow.com/questions/11626761/hibernate-criteria-query-to-get-specific-columns. Last accessed March 16, 2016.
[5] 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.
[6] Making a onetoone-relation lazy. http://stackoverflow.com/questions/1444227/making-a-onetoone-relation-lazy. Last accessed March 16, 2016.
[7] mysql - how many columns is too many? http://stackoverflow.com/questions/3184478/how-many-columns-is-too-many-columns. Last accessed March 16, 2016.
[8] NHibernate update on single property updates all properties in sql. http://stackoverflow.com/questions/813240/nhibernate-update-on-single-property-updates-all\-properties-in-sql. Last accessed March 16, 2016.
[9] Why in JPA Hibernate update query ; all attributes get update in sql. http://stackoverflow.com/questions/10315377/why-in-jpa-hibernate-update-query-all-attributes\-get-update-in-sql. Last accessed March 16, 2016.
[10] D. Barry and T. Stanienda. Solving the java object storage problem. *Computer*, 31(11):33–40, Nov. 1998.
[11] R. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
[12] I. T. Bowman and K. Salem. Optimization of query streams using semantic prefetching. *ACM Trans. Database Syst.*, 30(4):1056–1101, Dec. 2005.
[13] S. Chaudhuri, V. Narasayya, and M. Syamala. Bridging the application and DBMS profiling divide for database application developers. In *VLDB*, pages 1039–1042. Very Large Data Bases Endowment Inc., 2007.
[14] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 375–386, 2011.
[15] T.-H. Chen, S. Weiyi, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. 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, 2014.
[16] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.
[17] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 International Conference on Management of Data*, SIGMOD '13, pages 931–942, 2014.
[18] A. Cheung, A. Solar-Lezama, and S. Madden. 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, 2013.
[19] A. E. Chis. 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, 2008.
[20] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. L. Erlbaum Associates, 1988.
[21] B. Commerce. Broadleaf commerce. http://www.broadleafcommerce.org/. Last accessed March 16, 2016.
[22] J. Community. Hibernate. http://www.hibernate.org/. Last accessed March 16, 2016.
[23] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1403–1414, 2009.
[24] Django. Specifying which fields to save. https://docs.djangoproject.com/en/dev/ref/models/instances/#specifying-which-fields-to-save. Last accessed March 16, 2016.
[25] J. Dubois. 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-5/. Last accessed March 16, 2016.
[26] EclipseLink. Eclipselink documentation. http://www.eclipse.org/eclipselink/documentation/2.5/solutions/migrhib002.htm. Last accessed March 16, 2016.
[27] EclipseLink. Eclipselink documentation. http://eclipse.org/eclipselink/documentation/2.4/concepts/descriptors002.htm. Last accessed March 16, 2016.
[28] A. S. Foundation. Apache openjpa. http://openjpa.apache.org/. Last accessed March 16, 2016.
[29] E. Foundation. Eclipse java development tools. https://eclipse.org/jdt/. Last accessed March 16, 2016.
[30] E. Foundation. Eclipselink jpa 2.1. https://wiki.eclipse.org/EclipseLink/Release/2.5/JPA21#Entity_Graphs. Last accessed May 16, 2016.
[31] T. E. Foundation. Aspectj. http://eclipse.org/aspectj/. Last accessed March 16, 2016.
[32] T. E. Foundation. Eclipselink. http://www.eclipse.org/eclipselink/. Last accessed March 16, 2016.
[33] A. Georges, D. Buytaert, and L. Eeckhout. 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, 2007.
[34] M. Grechanik, C. Fu, and Q. Xie. 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, 2012.
[35] M. Grechanik, B. Hossain, and U. Buy. 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, 2013.

[36] M. Grechanik, B. M. M. Hossain, U. Buy, and H. Wang. Preventing database deadlocks in applications. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 356–366, 2013.
[37] IBM. Websphere. http://www-01.ibm.com/software/ca/en/websphere/. Last accessed March 16, 2016.
[38] Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of 2008 IEEE International Conference on Software Maintenance*, pages 307–316, Sept 2008.
[39] R. Johnson. J2EE development frameworks. *Computer*, 38(1):107–110, 2005.
[40] M. Jovic, A. Adamoli, and M. Hauswirth. 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, 2011.
[41] T. Kalibera and R. Jones. marking in reasonable timerigorous benchmarking in reasonable time. In *Proceedings of the 2013 international symposium on International symposium on memory management*, ISMM '13, pages 63–74, 2013.
[42] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. Systematic review: A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.*, 49(11-12):1073–1086, Nov. 2007.
[43] M. Keith and M. Schincariol. *Pro JPA 2: Mastering the Javał Persistence API*. Apresspod Series. Apress, 2009.
[44] M. Keith and R. Stafford. Exposing the orm cache. *Queue*, 6(3):38–47, May 2008.
[45] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002.
[46] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988.
[47] N. Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, Aug. 2000.
[48] O. S. Ltd. Jpa performance benchmark. http://www.jpab.org/All/All/All.html. Last accessed March 16, 2016.
[49] C. McDonald. 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.
[50] L. Meurice and A. Cleve. Dahlia: A visual analyzer of database schema evolution. In *Proceedings of CSMR-WCRE 14'*, 2014.
[51] D. Moore, G. MacCabe, and B. Craig. *Introduction to the Practice of Statistics*. W.H. Freeman and Company, 2009.
[52] S. Nakagawa and I. C. Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, 82:591–605, 2007.
[53] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 2015 International Conference on Software Engineering*, ICSE '15, 2015.
[54] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, 2013.
[55] R. on Rails. 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.
[56] T. Parsons and J. Murphy. 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, 2004.
[57] S. PetClinic. Petclinic. https://github.com/SpringSource/spring-petclinic/. Last accessed March 16, 2016.
[58] D. Qiu, B. Li, and Z. Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of ESEC/FSE 13'*, pages 125–135, 2013.
[59] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 133–144, 2012.
[60] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. 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, 2008.
[61] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. 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, 2013.
[62] SpringSource. Spring framework. www.springsource.org/. Last accessed March 16, 2016.
[63] J. Sutherland. How to improve JPA performance by 1,825%. http://java-persistence-performance.blogspot.ca/2011/06/how-to-improve-jpa-performance-by-1825.html. Last accessed March 16, 2016.
[64] J. Sutherland and D. Clarke. *Java Persistence*. Wikibooks, 2013.
[65] A. van Hoorn, M. Rohr, and W. Hasselbring. 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. Springer Berlin Heidelberg, 2008.
[66] X. Xiao, S. Han, D. Zhang, and T. Xie. 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, 2013.
[67] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 174–186, 2010.
[68] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. 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, 2010.
[69] P. Zaitsev, V. Tkachenko, J. Zawodny, A. Lentz, and D. Balling. *High Performance MySQL: Optimization, Backups, Replication, and More*. O'Reilly Media, 2008.