Detecting Problems in the Database Access Code of Large Scale Systems

An Industrial Experience Report

Tse-Hsun Chen
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
tsehsun@cs.queensu.ca

Weiyi Shang
Concordia University
Quebec, Canada
shang@encs.concordia.ca

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
ahmed@cs.queensu.ca

Mohamed Nasser
BlackBerry, Canada

Parminder Flora
BlackBerry, Canada

ABSTRACT

Database management systems (DBMSs) are one of the most important components in modern large-scale systems. Thus, it is important for developers to write code that can access DBMS correctly and efficiently. Since the behaviour of database access code can sometimes be a blackbox for developers, writing good test cases to capture problems in database access code can be very difficult. In addition to testing, static bug detection tools are often used to detect problems in the code. However, existing bug detection tools usually fail to detect functional and performance problems in the database access code. In this paper, we document our industrial experience over the past few years on finding bug patterns of database access code, implementing a bug detection tool, and integrating the tool into daily practice. We discuss the challenges that we encountered and the day-to-day lessons that we learned during integrating our tool into the development processes. Since most systems nowadays are leveraging frameworks, we also provide a detailed discussion of five framework-specific database access bug 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 bug patterns and associated detectors.

1. INTRODUCTION

Due to the emergence of cloud computing and big data applications, modern software systems 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 systems, 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 locate problems (e.g., bugs) 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 bug detection tools, such as FindBugs [14] and PMD [27], to provide a complete coverage of the entire system. However, these bug detection tools usually only provide patterns for detecting general code bugs, but cannot detect domain-specific bug patterns that are associated with accessing databases. For example, modern systems 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 bug patterns. In addition, existing static bug detection tools usually rely on scanning the binary files, but many database access bug patterns that we see in practice require parsing specialized annotations in the code or analyzing external SQL scripts.

In our prior research [8], we implemented a prototype tool to detect two database access bug patterns in collaboration with industry. Our research-based domain-specific static bug 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 system. We 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 bug detection tool. In this paper, we document and discuss the challenge and lessons learned. We believe that our experiences can help other researchers improve bug detection tools and ensure a smoother adoption process of their tools in practice.

This paper also presents five framework-specific database access bug patterns that we observed while working on several industrial systems. Our goal is to give readers concrete examples of framework-specific bug patterns. Since most modern systems are leveraging frameworks, framework-specific bug patterns can have a large impact in practice. Hence, we hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language bug patterns and associated detectors.

© 2016 ACM. ISBN 978-1-4503-4205-6.
DOI: 10.1145/1235

Table 1: A list of popular static bug detection tools.

Tool	Focused bug types	Input	Language	Open sourced
FindBugs [14]	General	Binary	Java	Yes
Error Prone [13]	General	Binary	Java	Yes
Infer [10]	General	Binary	Multiple	Yes
PMD [27]	General	Source	Multiple	Yes
Coverity [9]	General	Source	Multiple	No
AppScan [15]	Security	Source	Multiple	No

The main contributions of this paper are:

- We find that most of the existing static bug detection tools are not able to detect bugs that are associated with database access code — highlighting the need for more specialized bug detection tools for domain-specific problems.
- We provide an experience report that discusses the lessons that we learned on discovering, locating, and detecting the bug patterns, and the challenges that we encountered when adopting our bug detection tool in practice.
- We provide detailed documentation on the root causes and impact of five database access bug patterns that we have seen in the studied industrial systems over the past few years.

Paper Organization. Section 2 surveys related work. Section 3 discusses the background story and the technologies that are related to the studied database access bug patterns. Section 4 discusses the challenges and lessons that we learned when integrating our tool in practice. Section 5 provides a detail discussion of the five studied database access bug patterns. Finally, Section 6 concludes the paper.

2. RELATED WORK

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

2.1 Static Bug Detection Tools

Table 1 shows a list of popular static bug detection tools. FindBugs [14] is an open source static Java bug detection tool that is widely used. FindBugs scans Java binaries to detect general bugs, bad coding styles, and some security problems. PMD [27] is a static source code analyzer that detect potential problems in the code using pre-defined rules related to general coding problems (e.g., empty code blocks and bad code design). Error Prone [13] is a code analysis tool developed by Google, where the 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 related to coding errors such as comparing arrays using "==". Infer [10] is a static bug 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 [9] is a commercial static bug detection tool, which looks for different kinds of bug patterns in various programming languages. Finally, AppScan [15] is a static bug detection tool developed by IBM and is specialized in detecting security bugs.

Although the above-mentioned static bug detection tools are widely used and are able to detect many kinds of bugs, none of them have emphasis on detecting domain-specific bugs, such as bugs related to performance or database. However, as recent studies show [8,17,25,26], such bugs also have significant impact on software quality, and developers are interested in detecting those bugs.

In this paper, we discuss our experience on discovering and detecting database access bug patterns, which the above-mentioned tools do not detect. As modern systems are relying more heavily on DBMSs, detecting database access bugs can significantly help improve the user experience and software quality. We document some functional and non-functional (i.e., performance-related) database access bug patterns that we see in practice, and provide the challenges that we encountered and the lessons that we learned when adopting our static bug detection tool into practice.

2.2 Integrating Code Analysis Research Tools into Practice

In addition to this paper, there are some prior studies discussing the challenges associated with adopting static bug detection research tool into practice. Johnson *et al.* [18] interview 20 developers regarding the challenges that they see when using static bug detection tools. Johnson *et al.* find that tool configuration (e.g., filtering mechanism), integration with development workflow, and report formatting affect developers' willingness to use a static bug detection tool. Ayewah *et al.* [6] 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.* [24] discuss how they use an online portal to help improve the adoption of static bug detection tools at IBM. The portal provides cloud-based code scanning and allows developers to communicate by adding discussion to the static bug detection reports. Smith *et al.* [29] conduct a user study on the questions that developers ask when using static bug detection based security tools. They found that the security tools should provide better reporting systems to help developers locate the problems.

In this paper, we focus on database access bug patterns, which have different characteristics than general bug patterns. For example, due to the differences in the nature of the accessed database tables (e.g., size), some detected bug patterns may be more severe in practice. Since adopting research results in practice can be very challenging [22], we discuss the challenges and lessons that we learned when adopting our static bug detection tool in practice.

3. BACKGROUND

In this section, we briefly discuss the industrial systems that we use, and the background story for creating a tool to detect database access bug patterns.

Studied Systems. In this paper, we document the database access bug patterns that we see in industrial systems over the past years. Due to non-disclosure agreement (NDA), we cannot give the exact details of the systems. However, the industrial systems 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 systems often use for accessing DBMS and managing database transactions.

That there are a slew of similar technologies used in prac-

tice 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 these two technologies below because of their popularity and our need to provide concrete examples throughout the paper, so the reader can better grasp the raised concerns and the documented patterns.

Accessing DBMS Using Hibernate. Hibernate is a Java-based object-relational mapping (ORM) framework that is used for accessing the DBMS. Hibernate is the most popular Java ORM used in industry. A recent survey [31] finds that that among 2,164 surveyed Java developers, 67.5% use Hibernate for accessing the DBMS. ORM frameworks automatically map objects in object-oriented languages to records in the DBMS, and ORM frameworks can automatically translate object manipulations to database operations. Thus, ORM frameworks have become very popular [19] in the industry, since using ORM can significantly reduce maintenance efforts and the amount of boilerplate code that needs to be written to interface with modern DBMSs [20].

Figure 1 shows an example of using Hibernate to configure a class as a database entity class (a class that is mapped to a table in the DBMS). Developers can add annotations such as **@Entity** to specify a class as a database entity class, which is then mapped to a table in the DBMS (specified using the **@Table**). Developers can add **@Id** to map an instance variable to the primary key, and use **@Column** to specify which column an instance variable is mapped to. Developers can also specify the relationships between database entity classes. In this example, there is a one-to-many relationship between User and Group, which means that a user may belong to one or many groups. Developers can configure how the associated objects should be fetched. An *EAGER* fetch type means that the associated objects will always be retrieved regardless whether they will be used or not. For example, if fetch type is *EAGER*, fetching Group will always also fetch all Users in that group. On the other hand, a fetch type of *LAZY* means that the associated objects will only be fetched whenever they are used in the code.

After configuring a class as a database entity class, developers can use code such as:

```
User u1 = new User("Peter");
save(u1);
User u2 = find(User.class, id);
```

to insert a new user to the DBMS, or retrieve a user by id from the DBMS. Thus, using Hibernate significantly abstracts the details of the underlying database access.

In our experience, we have seen a number of database access bug patterns that are related to Hibernate. Some bug patterns occur because of Hibernate's database abstraction. Some developers may not notice that the code that they write may access the DBMS, and thus end up in writing inefficient database access code [8]. We also see bug patterns that are caused by system evolution. For example, using an *EAGER* fetch may work better for some use cases, but may cause performance problems in other newly introduced use cases. Thus, automatically identifying such suboptimal uses of configurations is important, as systems constantly evolve.

Transaction Management Using Spring. Spring [30] is a widely used framework for database transaction management, based on an aspect-oriented approach. A recent survey [31] 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:

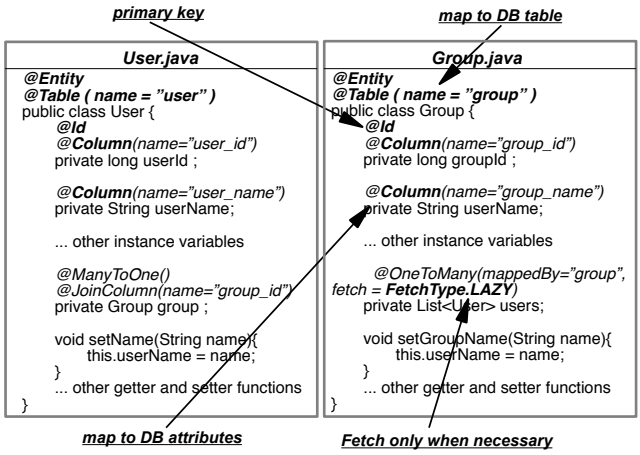

Figure 1: An example of configuring database entity classes using Hibernate.

```
@Transactional
public void performBusinessTransaction(){
        ...
}
```

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 system.

In practice, we see some database access bug 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).

4. CHALLENGES AND LESSONS LEARNED

We encountered many bugs that are associated with accessing a DBMS when developing large-scale industrial systems. In our prior work [8], we derived some bug patterns based on such bugs, and implemented a prototype static bug 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

bug patterns that the developers have seen over the years in the field. Based on our experience and from developers' feedback, these database access bug patterns can have significant impact on the system quality. As these database access bug patterns cannot be detected using readily available static bug detection tools such as FindBugs or PMD, the first author implemented the detection algorithms and integrated them into our *DBChecker* tool. However, although developers see value in our database access bug 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 bug 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 bug detection research and practice.

Below, we provide detailed discussions on the challenges that we encountered when adopting our tool to detect database access bug 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 bug detection tools is that these tools usually report a large number of problems that overwhelm the developers [18, 28], and our tool is not an exception. However, we found that not all of the detected problems 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 bug detection tool in practice. We found that many detected performance bug patterns are true bugs, but their impact may be too small to be relevant.

Challenge Impact. Showing all the detected problems 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 problems. So it is important to highlight the problems that have the highest impact. Therefore, helping developers make fast decision on whether a detected problem 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 problems may not have the same impact, even though the problems belong to the same pattern. For example, a detected problem that is related to a frequently accessed database table can have a larger impact in practice, compared to problems 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 a problem [12]. 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 Problems. Grouping the detected problems allows developers to allocate more resources to important components of the system. Figure 2 shows an example detection report of the *nested transaction* bug pattern (c.f. Section 5.2.1). We group the detected problems 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 the same package and affected by the same problem). In order to identify whether the detected problem is real and has a sizeable impact in a timely manner, our tool also recommends the experts that should investigate the problem, based on the developer who last modified the method, in which the detected problem was found.

> *To help developers allocate quality assurance effort, we group the detected problems according to their locations in the code.*

Prioritizing Detected Problems. We found that some of the studied database access bug patterns may have varying severity in different use cases, especially for performance related bug patterns. For example, if a bug pattern is related to relationships between two database tables, a many-to-many relationship would be more severe than one-to-one [8]. In the case of eagerly fetching data from the DBMS, a one-to-many relationship between two tables, e.g., user and group in Figure 1, 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 problem becomes more severe compared to the same pattern but with a one-to-one relationship. Hence, it is important to prioritize the detected problems according to their potential severity to reduce the inspection effort of developers. We also found 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 problems 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 problems that are related to those tables.

> *To help developers allocate quality assurance efforts, we prioritize the detected problems according to their potential severity.*

Characterizing the Detected Problems. In order to improve the readability of our report and help developers understand the detected problems faster, we provide a detailed breakdown of each detected problem in the report. Figure 2 shows an example detection report of the *nested transaction* bug pattern. For each detected problem, 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

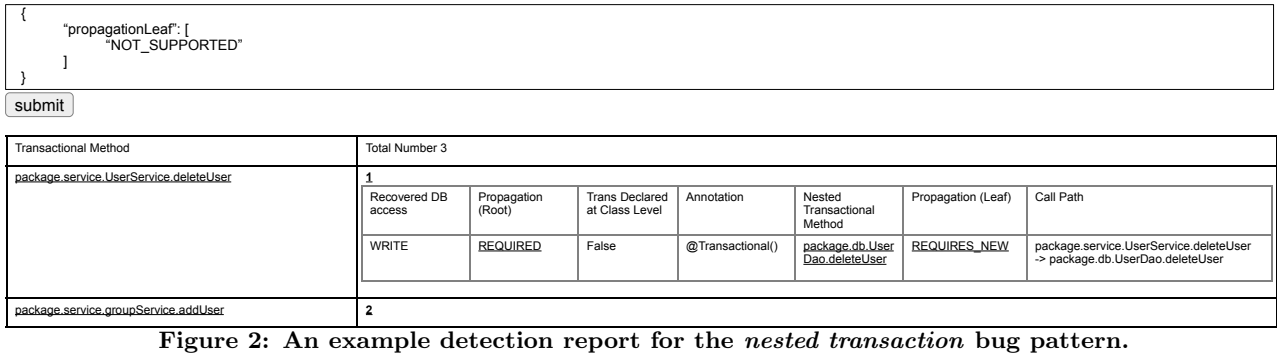

Figure 2: An example detection report for the *nested transaction* bug pattern.

the root transactional method to the nested transactional method. We provide similar breakdowns of each detect problems for other database access bug patterns. In short, we found that by providing a detailed breakdown of each bug pattern 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 problem 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.* [18], developers usually want more customizability of static bug 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 bug 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 problem is useful for hiding detected database access bug patterns that developers have already verified, or hiding bug patterns that are less interesting (e.g., the bug patterns have minor impact or the component with detected bug 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 2, developers can decide that all detected problems 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 problem in the report, *all detected problems* 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 problems according to the transactional method (e.g., deleteUser), then only that particular problem would be hidden. Developers can also use the text area (i.e., the form with a submit button in Figure 2) to see and to update their previous decisions.

In our experience, integrating developers' feedback on detected problems can help developers prioritize their resources.

C2: Giving Developers Rapid Feedback

Challenge Description. We found that it is difficult to ask every developer to run static bug detection tools in his/her own local environment. A similar challenge was previously encountered by Shen *et al.* while using other static bug detection tools [28]. Setting up and running the tools may interrupt developers' common workflow. However, it is important to provide developers with prompt alerts about new problems 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 problems. 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 problems. Developers may forget about the details of the code that caused the problems, which makes it even more difficult to fix the detected problems. Moreover, since there may be new code that is dependent on the detected problems, fixing the detected problems 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 [7] is a common development process for ensuring the quality of the system, where development teams continuously generate products (newer versions of a system) that are reliable for releasing in short cycles. For example, Facebook releases new version of its system into production twice a day, and Amazon makes changes to its production systems 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 systems once a day, and generate a report of all the detected problems, as well as the new problems that are introduced since its last run. Since there is not usually a large amount of new code is added since the last run, developers only need to examine a small number of newly introduced problems. 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 bug detection tool in the Continuous Delivery process can help increase developers' interest in the detection results and allow prompt attention to the detected problems.

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

C3: Maintaining Developers' Interest in the Detection Results

Challenge Description. We found that developers may lose interests in the static bug detection results if the detected problems are not related to the components that are under active development, or if the detected problems 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 problems that might not have an impact in the field yet.

Challenge Impact. Static bug detection tools are only useful if the developers are willing to investigate the detected problems 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 bug detection tools, it is important to have developers involved in tool development to some extent. In our previous study [8], we implemented our static bug detection tool and walked developers through the bugs that we found. The developers were not only interested in the problems that we detected, but they were also interested in how the tool was developed and whether the tool can be extended to detect other bug patterns. Now we sometimes receive requests from developers to implement detectors for new bug 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 problems related to the new bug patterns that they asked us to develop, since developers are still actively working on fixing those problems.

We found that having developers involved in the development and discussion of the static bug detection tool can increase developers' interest and motivation to fix the detected problems.

C4: Communicating the Problems with Developers

Challenge Description. As systems 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 problem to speed up the bug fixing process.

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

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

We hosted several "static bug 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 problems. One of the benefits of hosting such workshops is to learn more patterns from developers' experiences. In fact, we found most of the studied bug patterns in this paper through interacting with developers in the workshops.

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

We found that it is necessary to educate developers about the root causes and the possible impact of the detected problems to increase developers' awareness of the severity of the detected problems. In addition, an effective bug triaging policy is needed to react rapidly to the detected problems.

5. DATABASE ACCESS BUG PATTERNS

5.1 The Need for Framework-Specific and Non-General Bug Patterns

The detection algorithms that we use to detect the studied database access bug patterns are straightforward, but knowing the bug patterns in the first place requires extensive domain knowledge. As systems become more complex, developers start to leverage different frameworks and technologies during development. There may be many new kinds of bug patterns that are related to these frameworks, but since these bug patterns are not available in most existing static bug detection tools, developers are left in the dark. As an example, a recent study [31] 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 bug detection tools, there is only one Hibernate-related bug pattern (which is related to SQL injection) in Coverity. Therefore, finding bug patterns that are more specific can further help improve system quality significantly. Due to the wide rise of frameworks throughout industry, in the following subsection, we discuss the framework-specific database access bug patterns that we have seen in practice. Our goal of presenting these patterns is to give readers concrete examples of framework-specific bug patterns. Hence, we hope to encourage further research efforts on framework-specific detectors, instead of the current research focus on general programming language bug patterns and associated detectors. We feel future studies should also consider detecting bug patterns that may be more specific to certain frameworks but have a significant impact in most industrial systems.

5.2 Studied Bug Patterns

After our prototype [8] was adopted in practice, we received active feedback from developers, and they showed enormous interest in the detected problems. After active discussion and cooperation with the developers, we found five new database access bug patterns. These bugs have caused both functional and non-functional problems in the systems, and some problems were difficult to capture. Unfortunately, current static bug detection tools such as FindBugs or PMD fail to capture most of these bug patterns. As a result, we improved our *DBChecker* tool to further detect these five database access bug patterns.

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

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

Description. A detailed description of the database access bug pattern.

Example. An example of the bug pattern.

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

Possible solutions. We discuss possible solutions to resolve the bug pattern.

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

5.2.1 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 3). 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 system 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:

```
Class A{
        @Transactional(timeout = 300ms)
        public User updateUserById(int id) {
                ...
                notifyServer();
                ...
        }
}
Class B{
        @Transactional(REQUIRES_NEW)
        public void notifyServer(){
                ...
        }
}
```

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 [16], such as blocking DBMS connection or deadlock. Thus, it is important to notify developers about *nested transaction*, and manually investigate if the detected bug 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 system, 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*.

5.2.2 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:

```
Class A{
        @Transactional(timeout = 300ms)
        public User updateUserById(int id) {
                ...
                notifyServer();
        }
        @Transactional(REQUIRES_NEW)
```

```
        public void notifyServer(){
                ...
        }
}
```

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 [3, 4]. Detecting such bug 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 a system 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 system, 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(REQUIRES_NEW)** and both annotated methods are defined in the same class.

5.2.3 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 [23], 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:

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

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 [1, 2]. 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 system, 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 problem as a warning.

5.2.4 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:

```
Class User{
        @Id
        @SequenceGenerator(sequenceName=
                "user_seq")
        @Column(name="user_id")
        private int id;
        ...
}

user_schema.sql
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 [21]. However, most existing static bug detection

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.

5.2.5 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:

```
group.getUserList().clear();
group.addUser(user);
update(group);
```

resulting SQL queries:

```
INSERT into User values ...
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 [5, 11], 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 systems.

Detection approach. To detect *incorrect SQL order*, we perform control flow analysis on the system 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. CONCLUSION

In this paper, we provide an experience report on the challenges and lessons that we learned when adopting our static bug detection tool in practice. Our static bug detection tool focuses on detecting database access bug patterns that existing static bug detection tools cannot detect. Since these database access bug patterns are different from general code bug patterns in many aspects, we discuss how we help developers prioritize detection results and what is needed to detect additional specialized bug 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 bug patterns that we have observed in large-scale industrial systems over the past years. We believe that our findings can help researchers create static bug 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 systems nowadays are leveraging frameworks instead of being built with basic programming constructs (the main focus of much of the current research in static bug detectors).

Acknowledgments

We are grateful to BlackBerry for providing access to many of the enterprise systems that we 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.

7. REFERENCES

[1] Spring @Transactional read-only propagation. http://stackoverflow.com/questions/1614139/spring-transactional-read-only-propagation, 2009. Last accessed 15 Feb 2016.

[2] Spring transaction readonly. http://stackoverflow.com/questions/2562865/spring-transaction-readonly, 2010. Last accessed 15 Feb 2016.

[3] Starting new transaction in spring bean. http://stackoverflow.com/questions/3037006/starting-new-transaction-in-spring-bean, 2010. Last accessed 15 Feb 2016.

[4] Spring transaction: requires_new beharivour. http://stackoverflow.com/questions/22927763/spring-transaction-requires-new-beharivour, 2014. Last accessed 15 Feb 2016.

[5] 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, 2015. Last accessed 15 Feb 2016.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. 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, 2007.

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

[8] T.-H. Chen, W. Shang, 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 2014, pages 1001–1012, 2014.

[9] Coverity. Coverity code advisor. http://www.coverity.com/, 2016. Last accessed 15 Feb 2016.

[10] Facebook. Infer. http://fbinfer.com/, 2016. Last accessed 15 Feb 2016.

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

[12] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. 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, 2007.

[13] Google. Error prone. http://errorprone.info/, 2016. Last accessed 15 Feb 2016.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[15] IBM. Security appscan source. http://www-03.ibm.com/software/products/en/appscan-source, 2016. Last accessed 15 Feb 2016.

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

[17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, 2012.

[18] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. 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, 2013.

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

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

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, Mar. 2006.

[22] D. Lo, N. Nagappan, and T. Zimmermann. 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, 2015.

[23] V. MIHALCEA. A beginner's guide to JPA/Hibernate flush strategies. http://vladmihalcea.com/2014/08/07/a-beginners-guide-to-jpahibernate-flush-strategies/, 2014. Last accessed 15 Feb 2016.

[24] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 99–108, 2010.

[25] 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, pages 902–912, 2015.

[26] 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.

[27] PMD. Pmd. https://pmd.github.io/, 2016. Last accessed 15 Feb 2016.

[28] H. Shen, J. Fang, and J. Zhao. 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, 2011.

[29] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. 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, 2015.

[30] SpringSource. Spring framework. www.springsource.org/, 2013. Last accessed 15 Feb 2016.

[31] ZeroturnAround. Java tools and technologies landscape for 2015. http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/, 2014. Last accessed 15 Feb 2016.