Improving the Quality of Large-Scale Database-Centric Software Systems by Analyzing Database Access Code

Tse-Hsun Chen
supervised by Ahmed E. Hassan
Expected graduation date: Summer of 2016

Software Analysis and Intelligence Lab (SAIL), Queens University, Canada
{tsehsun, ahmed}@cs.queensu.ca

***Abstract*—Due to the emergence of cloud computing and big data applications, modern software systems are becoming more dependent on the underlying database management systems (DBMSs) for data integrity and management. Since DBMSs are very complex and each technology has some implementation-specific differences, DBMSs are usually used as black boxes by software developers, which allow better adaption and abstraction of different database technologies. For example, Object-Relational Mapping (ORM) is one of the most popular database abstraction approaches that developers use. Using ORM, objects in Object-Oriented languages are mapped to records in the DBMS, and object manipulations are automatically translated to SQL queries. Despite ORM's convenience, there exists impedance mismatches between the Object-Oriented paradigm and the relational DBMSs. Such impedance mismatches may result in developers writing inefficiently and/or incorrectly database access code. Thus, this thesis proposes several approaches to improve the quality of database-centric software systems by looking at the application source code. We focus on troubleshooting and detecting inefficient (i.e., performance problems) and incorrect (i.e., functional problems) database accesses in the source code, and we prioritize the detected problems based on severity. Through case studies on large commercial and open source systems, we plan to demonstrate the value of improving the quality of database-centric software systems from a new perspective – helping developers access the database more efficiently and accurately.**

I. Introduction

Due to the emergence of cloud computing and big data applications (e.g., Amazon, BlackBerry, and Google), modern software systems are becoming more dependent on the underlying database management systems (DBMSs) for data integrity and management. These large-scale database-centric software systems pose new challenges for the database and software engineering field, since these systems need to be responsive while being able to support millions of concurrent users at the same time.

DBMSs are one of the core components of database-centric systems. Developers often store all user data in DBMSs to provide better scalability and maintainability. Although DBMSs are usually fairly optimized in terms of performance and data management, how developers control and communicate with the DBMS has a significant impact on the quality of database-centric software systems.

Since managing the data consistency between the source code and the DBMS is a difficult task, especially for complex large-scale systems, technologies are leveraged to ease the data access. For example, developers often employ Object-Relation Mapping (ORM) frameworks to provide a conceptual abstraction between objects in Object-Oriented Languages and records in the underlying DBMS. Using ORM, changes on the object states are automatically propagated to the corresponding records in the DBMS. These abstraction frameworks significantly reduce the amount of code that developers need to write [1], [2]. However, due to the impedance mismatch between the Object-Oriented paradigm and the relational model, developers may write code that access the DBMS inefficiently (i.e., causes performance problems) and/or incorrectly (i.e., causes functional problems).

Recent studies (e.g., [3], [4], [5]) have proposed various frameworks on top of database-access abstraction layers (e.g., ORM) to transform and optimize the automatically generated SQLs. However, a potential problem with these approaches is that the system may experience higher overheads and become harder to debug due to the extra layer of complexity. Thus, in this thesis, we propose several new approaches to detect potential problems in the application source code, and we rank the problems according to their severity. Our approaches add little to no overheads to the system, and developers can allocate their effort effectively according to our results. We will demonstrate our proposed contributions using both open source and commercial systems.

Our research makes the following contributions:

- We have proposed approaches using static and dynamic analysis to detect problematic database access code.
- We have proposed an approach to help developers locate data mismatches between the needed data in the application code and the requested data by ORM.
- We have proposed a non-intrusive approach to monitor runtime data interactions.
- We have proposed an approach to understand and detect

transaction management bugs in systems that are implemented using ORM.
- Part of our initial work is adapted by BlackBerry to ensure the performance and quality of their large-scale database-centric systems.

Paper Organization. Section II surveys related work, and Section III outlines our research hypothesis. Section IV discusses our exploratory study on the maintenance of database abstraction code. Section V introduces the research problems and our proposed approaches for improving the quality of database-centric systems. Finally Section VI concludes the paper.

II. Related Work

In this section, we discuss related research that aims to help developers improve the quality of database-centric systems.

A. Optimizing the Performance of Generated Queries

Smith and Williams [6] first document the problem and possible solutions of a number of database-related performance anti-patterns. They discuss a pattern called *Empty Semi Trucks*, which occurs when a large number of excessive query calls (e.g., select, insert, update, or delete) is sent to the DBMS for a given task. Chen *et al*. [7] implement a static analysis framework, which automatically detects performance anti-patterns that are similar to *Empty Semi Trucks*. Smith and Williams [6] do not provide any detection approaches for *Empty Semi Trucks* and they only discuss non-ORM systems. Chen *et al*. apply static analysis to detect ORM DBMS calls within loops, which result in a large number of repetitive SQL queries. In addition, Chen *et al*. [7] apply static analysis to detect data that is retrieved from a DBMS, but never used by the system. Cheung *et al*. [4] optimize the system performance using SQL query synthesis. SQL query synthesis aims to generate SQL queries automatically according to some predefineded constraints. The approach that Cheung *et al*. developed is based on the pre- and post-conditions of methods that access the DBMS. Pre-conditions are the conditions that must always hold before executing a method, and post-conditions are the conditions that must always hold after executing a method. By generating optimized queries that satisfy the pre- and post-conditions, Cheung *et al*. transform SQL query into a more efficient form. Chaudhuri *et al*. [8] propose an approach to link functions with the SQLs that they generate. Their approach can help locate the root cause of problematic SQLs (i.e., the functions that generate the SQLs). Tamayo *et al*. [9] construct the program dependency graph of a database-centric system using dynamic information flow, and combine the information with the corresponding query operations to identify performance bottlenecks. Their tool may also be used to identify problems related to batching, SQL synchronization, repetitive SQL queries, and extra operations.

B. Reducing Query Transmission Latency

Cheung *et al*. [3] dynamically keep track of the SQL queries, and delay all the query operations (i.e., submission of queries and computations) as late as possible. Their approach reduces the data transmission overheads between the DBMS and a database-centric system by sending the queries to the DBMS in one large batch. Thus, the impact of round-trip time and network latency can be reduced. Chavan *et al*. [5] use static analysis to automatically transform source code to allow asynchronous query submission. Their approach allows the system to continue doing other computations while waiting for the results of queries to return from the DBMS. Thus, total system execution time for a task can be reduced.

C. Deadlock Analysis by Analyzing the Source Code

Grechanik *et al*. [10] develop an approach for detecting database deadlocks in database-centric systems. In another work, Grechanik *et al*. [11] combine dynamic and static analysis to prevent database deadlocks. They first discover SQL queries that belong in the same transaction, and derive a wait-for graph. If two transactions are executed in an order that may cause a deadlock, they postpone one of the transactions. Jula *et al*. [12] collect deadlock patterns, which are patterns that may lead to deadlocks. They derive the patterns using traces collected during system execution. Future executions that contain such patterns are avoided by rescheduling.

D. Improving Database Schema

Nijjar *et al*. [13] extract formal mathematical models from the database schema of Ruby on Rails applications, and look for errors in the models. In their follow-up work, Nijjar *et al*. [14] extract formal mathematical models from the database schema, and develop heuristics to discover anti-patterns in the schema. Their framework can then automatically propose solutions to correct the database schema.

From the survey, we discover that prior work usually requires adding some frameworks on top of ORM, or requires developers to have deep knowledge of the database. However, adding another layer of complex framework may bring extra overheads and make the system harder to debug. In addition, since ORM abstracts database accesses from developers, many of the prior approaches that depend on developer-written queries may not work. Hence, to ensure the quality of database access code, this thesis proposes a series of approaches to help detect and locate inefficient and/or incorrect database access code when using ORM.

III. RESEARCH HYPOTHESIS

A large amount of prior work [15], [16], [17], [18], [19] has advanced the performance and quality of DBMSs. However, software systems still suffer from database-related problems, and one of the major reasons is related to how developers access the DBMS [4], [6], [7], [20]. We believe that by helping developers write better database access code, we can significantly improve the quality of database-centric systems.

In this thesis, we leverage both static code analysis and dynamic analysis to uncover potential problems about how

developers access the DBMS. Our approaches differ from previous work in that they are usually introducing an extra layer between the system and the DBMS, which may increase the difficulty of debugging. Our underlying research hypothesis is:

Due to the impedance mismatch between the Object-Oriented paradigm and the relational model, developers may write code that access the DBMS inefficiently (i.e., causes performance problems) and/or incorrectly (i.e., causes functional problems). By detecting and locating such inefficient and incorrect code, we can significantly improve the quality of database-centric software systems.

IV. An Exploratory Study on the Maintenance Efforts of Database Abstraction Code

DBMSs are usually abstracted from developers through various database abstraction technologies such as ORM. Thus, developers may not be aware that the code they write may interact with the DBMS, or they may not know how changing the DBMS configurations will affect the code (or vice versa). As a pilot study, we conducted a study on the maintenance of database abstraction technologies (we focus on Java ORM due to its popularity [4]) using three open source systems (Broadleaf Commerce [21], Devproof [22], and JeeSite [23]) and one enterprise system. We find that ORM code is usually the core of the system (more files depend on ORM code), and ORM code is changed more frequent than regular code (115% – 179% more). Moreover, developers usually need to change ORM code to accommodate database schema changes (56% of all ORM code changes), and configure ORM to achieve better usability (36%) and performance (8%). In short, we find that even though ORM tries to abstract database accesses from developers, many ORM code changes are still bounded to the underlying DBMS. Thus, writing efficient and correct ORM code may significantly improve the quality of database-centric system.

We find that ORM code is usually the core of a system, and developers constantly maintain it. However, even though ORM tries to abstract database accesses, most ORM code changes are still bounded to the underlying database.

V. Our Proposed Approach

In this section, we discuss our proposed approaches to improve the quality of database-centric systems by finding problems in how developers access the DBMS. We present our approaches for detecting inefficient and incorrect database access code in three subsections. For each approach, we present the problem we wish to solve and a brief overview of our proposed approach, evaluation criteria, and preliminary results if available.

A. Our Approaches for Detecting Inefficient Database Access Code

Detecting Performance Anti-patterns for Systems Implemented Using ORM

Problem: Using ORM, object manipulations are automatically translated to SQL queries. Thus, developers may not be aware which source code snippet would result in database access nor the generated SQLs are efficient. As a result, Developers may not proactively optimize the ORM database access code to improve system performance.

Our proposed approach: We have proposed a framework to automatically detect two types of ORM performance anti-patterns [7]. We discovered the anti-patterns through empirically study and our industrial partner, BlackBerry. We leverage static code analysis to detect such patterns, and our framework can also automatically prioritize the detected instances of anti-patterns according to their severity. The two anti-patterns that our framework can detect are: one-by-one processing and excessive data. One-by-one processing happens when developers write database access code in a loop, which results in generating a large number of similar but repetitive SQLs (the ideal solution would be to execute the SQLs in batches) [6]. Such code may be harder to find due to the ORM abstraction. Excessive data is related to ORM configurations, where ORM eagerly fetch data from associated table (i.e., SQL joins), but the eagerly fetched data is never used in the application code (i.e., join is not necessary). More detailed descriptions of our approach can be found in [7].

Evaluation Criteria: We apply dynamic analysis to rank and prioritize the anti-pattern instances. Our studied systems are Broadleaf [21], Pet Clinic [24], and an industrial system. We apply an automated code transformation on the ORM generated SQLs to remove the anti-patterns, a similar approach by Jovic *et al*. [25]. Then, we compare the performance between the SQLs with and without the anti-patterns.

Preliminary Results: Our framework is able to detect hundreds of anti-pattern instances in our studied systems, and developers have confirmed that some of the detected instances are indeed the root cases of performance bugs. We use MySQL as our DBMS, and we conduct our experiment on two separate machines, one for sending SQL requests, and one for hosting the DBMS. We find that by removing the anti-patterns, the system performance (i.e., response time) can improve 35% on average in our studied systems [7]. In addition, our dynamic analysis can help reduce the false positive in our static analysis results, so developers can focus on fixing more severe problems first. Our framework is now being used at BlackBerry on a daily basis to help the company ensure the quality of their database-centric systems.

Detecting Mismatches Between the Needed Data and the Requested Data

Problem: When using ORM or other similar database abstraction technologies, developers do not need to worry about writing SQL queries. However, since these abstraction technologies are generic and they do not know what data is needed in the application code, the needed data in the application

code and the requested data by the abstraction technologies may be different. The mismatches can result in serious performance problems when, for example, the generated SQLs are constantly updating unchanged non-clustered indexed columns in a table or retrieving unused BLOB columns [26], [27]. Although different ORM providers may have different ways to resolve the mismatches, developers may not know what to look for and where to start in a large system.

Proposed Approach: We plan to propose an approach that can automatically detect and locate the mismatches between the needed data in the application code and the ORM requested data. Our approach is general and should be applicable to other technologies. Our approach contains three steps: 1) static analysis for finding data read/write functions in the application code; 2) dynamic analysis for detecting the needed data in the application code and the requested data by the generated SQLs; and 3) locating the mismatches in the code. For the static analysis part, we plan to identify all the functions that read or write to the variables, which store data returned by the DBMS. For example, if a function *A* reads a user's name, and the value of the user name variable was returned from the DBMS, then we mark *A* as a data-read function. For the dynamic analysis part, we plan to use binary code instrumentation to monitor the execution of the data-read and -write functions, and the SQLs being generated. By comparing the needed and requested database accesses, our approach can find that, for example, a BLOB column is retrieved from the DBMS, but the column data is never used in the code.

Evaluation Criteria: After collecting the execution information, we will compare the requested and the needed data within and across transactions to identify data mismatches in the application system. Finally, we plan to evaluate the performance impact of the detected data mismatches. We plan to first perform an automated code transformation on the collected problematic SQLs to generate SQLs that request/update only the needed/used data in the application code. Then, we plan to automatically execute the original SQLs (SQLs with data mismatches) and the transformed SQLs (SQLs without data mismatches) separately to compare the performance difference, and rank the code snippet with data mismatches according to the performance impact.

Preliminary Results: We conduct our experiment in the same settings mentioned above using Broadleaf [21], Pet Clinic [24], and an industrial system. We find that in our exercised workloads, each transaction may contain up to several mismatches. We find that resolving the mismatches can improve the system performance (i.e., response time) by 3–88%.

B. Our Approaches for Understanding System Execution

A Non-intrusive Profiler for Runtime Data Interactions

Problem: We discover several challenges when trying to implement and execute the above-mentioned approaches: 1) there is no direct link between the code and the generated SQLs; 2) even though we can recover the link using instrumentation, instrumentation framework gives significant overheads, which can be problematic for systems deployed in production and make our approach less usable. However, understanding data interactions during runtime can help identify performance bottlenecks and improve software quality [9], [8]. Thus, to overcome the challenges, it is important to propose a non-intrusive data interaction profiler which is easy to use and can be applied to systems deployed in production.

Proposed Approach: We plan to implement a framework that can automatically recover the runtime DBMS data interactions without instrumentations. We plan to first apply static analysis to discover the corresponding data access of system logs (e.g., Tomcat HTTP access logs), and then recover the runtime data interaction by analyzing the logs. For example, for SaaS systems that implement REST for their external APIs, our framework will scan all functions that handle the REST requests, and record possible data access in each function using static analysis. We will map each function to the corresponding HTTP access log, and then we will be able to recover the runtime data interactions by only analyzing HTTP access logs (we obtain possible data access associated with the log from the previous step). To improve the granularity and accuracy of our approach, we will analyze the identifier of each request and how the identifier is used in the request handling function. We will perform a data flow analysis to see if the identifier is being used as a selection or update criteria for database access, and thus we may map each request to the corresponding records in database tables. Although our proposed approach may miss some cases, it is easy to use and brings no overheads to the system.

Evaluation Criteria: We plan to compare the recovered data interactions (using logs and static analysis) with the actual data interactions (i.e., SQLs) that we collect using instrumentation. Then, we will be able to evaluate the accuracy of our proposed approach. We also plan to show some use cases of our approach, such as helping developers configure cache or finding database tables that may suffer from performance problems due to frequent locking.

C. Our Approaches for Detecting Incorrect Database Access Code

Identifying Incorrect Transaction Management Code

Problem: Transaction management is a difficult issue in large-scale systems, and such issue may become more difficult to detect when using abstraction technologies such as ORM. For example, due to the implementation of some ORM frameworks, updating and deleting the same record sequentially may cause exceptions [28]. In addition, many database-centric systems now use dependency injection (DI) frameworks [29], such as Spring [30], to manage transactions. As a result, managing transactions is not as straightforward as writing plain mutex, and may cause different kinds of multithreading issues related to DBMS.

Proposed Approach: We plan to conduct an empirical study on systems that are implemented using ORM and DI frameworks. We plan to investigate Github[1] bug reports, and sample

[1]https://github.com

the bug reports related to transaction management. Then, we will conduct a manual study on the root causes of these bugs, and examine how often the database access code is causing the problem. Finally, we will develop a framework that can detect these problems statically.

Evaluation Criteria: After implementing the framework, we plan to evaluate the false positive rate of the detection result by manual analysis and by reporting the detected problems to developers.

Preliminary Results: Our initial study shows that there exist different transaction management issues in such systems [28], [31]. For example, due to Hibernate's own unique key handling mechanism, incorrect insertion and deletion order may cause exceptions during system runtime [28].

VI. Conclusion

Our pilot empirical study finds that, although Object-Relational Mapping (ORM) frameworks abstract database accesses from developers, many of the ORM code changes are still related to the underlying DBMSs. As a result, we hypothesize that by improving the quality of the database access code that developers write, we can significantly improve the quality of database-centric systems that are implemented using such abstraction frameworks. We have proposed several approaches to detect inefficient ORM code, locate where to configure ORM code efficiently, monitor system execution non-intrusively, and discover transaction management problems.

ACKNOWLEDGEMENTS

The author thanks Dr. Ahmed E. Hassan and Dr. Weiyi Shang for their valuable comments on earlier drafts. The author is grateful to the Performance Engineering team at BlackBerry for the support. By working as a researcher in the team, the author has learnt to appreciate the current challenges and possible solutions to improve the quality of database-centric systems.

References

[1] D. Barry and T. Stanienda, "Solving the java object storage problem," *Computer*, vol. 31, no. 11, pp. 33–40, Nov. 1998.

[2] N. Leavitt, "Whatever happened to object-oriented databases?" *Computer*, vol. 33, no. 8, pp. 16–19, Aug. 2000.

[3] A. Cheung, A. Solar-Lezama, and S. Madden, "Sloth: Being lazy is a virtue (when issuing database queries)," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[4] ——, "Optimizing database-backed applications with query synthesis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, 2013, pp. 3–14.

[5] 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*, ser. ICDE '11, 2011, pp. 375–386.

[6] C. U. Smith and L. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Proceedings of the 2003 Computer Measurement Group Conference*, ser. CMG 2003, 2003.

[7] 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*, ser. ICSE, 2014.

[8] S. Chaudhuri, V. Narasayya, and M. Syamala, "Bridging the application and dbms profiling divide for database application developers," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1252–1262.

[9] J. M. Tamayo, A. Aiken, N. Bronson, and M. Sagiv, "Understanding the behavior of database operations under program control," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, 2012, pp. 983–996.

[10] 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*, ser. ICST '13, 2013, pp. 174–183.

[11] 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*, ser. ESEC/FSE 2013, 2013, pp. 356–366.

[12] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 295–308.

[13] J. Nijjar and T. Bultan, "Bounded verification of ruby on rails data models," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, New York, NY, USA, 2011, pp. 67–77.

[14] ——, "Data model property inference and repair," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 202–212.

[15] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "Progressive merge join: A generic and non-blocking sort-based join algorithm," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, 2002, pp. 299–310.

[16] M. F. Mokbel, M. Lu, and W. G. Aref, "Hash-merge join: A non-blocking join algorithm for producing fast and early join results," in *Proceedings of the 20th International Conference on Data Engineering*, ser. ICDE '04, 2004, pp. 251–.

[17] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, Mar. 1988.

[18] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 111–152, Jun. 1984.

[19] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[20] B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*, ser. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

[21] B. Commerce, "Broadleaf commerce," http://www.broadleafcommerce.org/, 2013.

[22] D. Portal, "Devproof portal," https://code.google.com/p/devproof/, 2014.

[23] ThinkGem, "JEEsite," http://jeesite.com/, 2014.

[24] S. PetClinic, "Petclinic," https://github.com/SpringSource/spring-petclinic/, 2013.

[25] 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*, ser. OOPSLA '11, 2011, pp. 155–170.

[26] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill Education, 2003.

[27] P. Zaitsev, V. Tkachenko, J. Zawodny, A. Lentz, and D. Balling, *High Performance MySQL: Optimization, Backups, Replication, and More*. O'Reilly Media, 2008.

[28] H. ORM, "wrong insert/delete order when updating record-set," https://hibernate.atlassian.net/browse/HHH-2801, 2014.

[29] D. R. Prasanna, *Dependency Injection*. Greenwich, CT, USA: Manning Publications Co., 2009.

[30] SpringSource, "Spring framework," www.springsource.org/, 2013.

[31] B. Commerce, "Fulfillmentoptionservice has transactional annotation at class level, which causes transactions to flush when reading something from the db from a controller," https://github.com/BroadleafCommerce/BroadleafCommerce/issues/1069, 2013.