Where Was This SQL Query Executed? A Static Concept Location Approach

Csaba Nagy, Loup Meurice, Anthony Cleve
PReCISE Research Center, University of Namur, Belgium
{csaba.nagy,loup.meurice,anthony.cleve}@unamur.be

***Abstract*—Concept location in software engineering is the process of identifying where a specific concept is implemented in the source code of a software system. It is a very common task performed by developers during development or maintenance, and many techniques have been studied by researchers to make it more efficient. However, most of the current techniques ignore the role of a database in the architecture of a system, which is also an important source of concepts or dependencies among them.**

In this paper, we present a concept location technique for data-intensive systems, as systems with at least one database server in their architecture which is intensively used by its clients. Specifically, we present a static technique for identifying the exact source code location from where a given SQL query was sent to the database. We evaluate our technique by collecting and locating SQL queries from testing scenarios of two open source Java systems under active development. With our technique, we are able to successfully identify the source of most of these queries.

***Keywords*—concept location; fault location; static analysis; data-intensive systems; SQL; JDBC; Hibernate**

I. Introduction

Before developers start working on a given change they always need to identify which parts of the source code implement the feature, and should be examined first. In practice, what they do is a *concept location* task (also known as *feature identification/location*) which is "*the process that identifies where a software system implements a specific concept*" [1]. The input of this process is a change request and the output is the location of the change, typically one or more methods in the source code. The boundaries here are vague, but recent studies define the concept location process as finding only one part of the concept implementation which is the starting point of a change, and the rest of the process (to identify the full extent of the change) will be dealt with impact analysis, say [2], [3].

There are many existing approaches for supporting developers in concept location tasks starting from simple pattern matching (so-called 'grep' techniques) to more sophisticated methods like IR-based techniques or dependency analysis [4]. However, existing approaches largely ignore the possible presence of a database in the architecture, which adds further sources of artifacts or dependencies among them to the process.

In this paper, we present a static analysis solution for a typical concept location problem for data-intensive systems: "*where was this query executed?*". In particular, we introduce a static analysis technique for identifying the source code location(s) where a given SQL query was sent to the database server.

Typical scenarios for this task are when queries need to be optimized for performance, or when they cause failures (e.g. a syntactic error or a deadlock issue). This task can become really hard as the complexity of a system grows, especially in the case of languages where queries are constructed dynamically (e.g. in Java applications using JDBC). For example, simple grep or code search techniques are not sufficient for systems where thousands of queries are constructed via string operations and methods deep in the call hierarchy. Moreover, persistence frameworks (like Hibernate) can hide the query construction from developers, further complicating the debugging of such issues.

Dynamic techniques exist to trace the query on the database or client side. However, dynamic analysis cannot help us in certain situations. Suppose that the user of the application experiences performance issues at the database; he identifies the query which causes the performance drop back in the log files of the database and sends a bug report. Since the problem occurred in the database and was reported by it (the client was not directly affected), we do not have a stack trace in the bug report. How can we determine where the query was prepared in the source code? We must reproduce everything exactly as the user did, which might prove impossible if we depend on the (possibly confidential) data stored in the database. In such situations, a static approach seems more appropriate.

II. Overview

Figure 1 shows the main steps of our concept location approach. The process starts with the developer who specifies the SQL query that he would like to find in the source code of an application. Then we analyze the source files including the database schema (e.g. in the format of SQL schema export). The result is a set of matching queries and the locations of the method invocations that send them to the database.

A. Data Access (Query) Extraction

In our analysis we start by determining all those locations in the source code where SQL queries are sent to the database, then resolve the set of potential queries for each location. There are two typical methods to send a SQL statement to a database server: it can be specified *explicitly* and executed by invoking some methods of APIs (e.g. ODBC or JDBC), or *implicitly*, by using higher level APIs, which hide the lower level communication with the database server from the developers (e.g. ORMs like Hibernate).

Here, we target two popular libraries from the world of Java: JDBC and Hibernate. With these libraries, one can access data tables as follows:

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

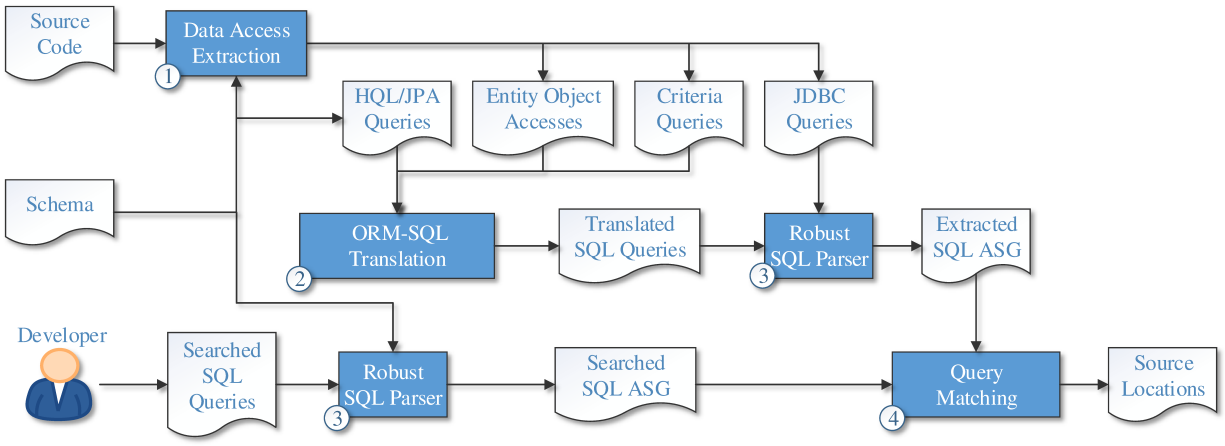

Fig. 1. Overview of the approach, where the main steps are in numbered boxes with their respective inputs and outputs

- Sending native SQL statements directly to the database server (e.g. by invoking `executeQuery()` in JDBC or `createSQLQuery()` in Hibernate)
- Constructing HQL or JPA queries as strings to query the database server (e.g. `createQuery()` in Hibernate)
- Using criteria queries
- Accessing entity objects (e.g `persist()`, `saveOrUpdate()`)

Our approach is currently able to handle the first two cases, when a query is specified via string operations and it is possible to extract the query string of a data access point. The success of this technique relies on the resolution of the values of string variables, for which we improved our previous study [5], and we trace back string variables following the control flow and call graph of the application. When a variable cannot be resolved statically, the query cannot be fully extracted and contains *unresolved query fragments*. As this case is very common, we design our concept location technique to be able to handle unresolved query fragments. For this step, it simply means that we use a special string '`@@null@@`' as a placeholder for such fragments.

```
160 List<Book> getBook(int code) {
161   String where="WHERE b.code=:code";
162   Query q = s.createQuery("b.title FROM Book b" +
          where);
163   q.setParameter("code", code);
164   List<Book> books = q.list();
165   return books;
166 }
```

Fig. 2. Example Hibernate HQL query construction

```
BookDAO.java(164): "b.title FROM Book b WHERE b.code
    =@@null@@"
```

Fig. 3. HQL query extracted from the code sample in Figure 2

```
BookDAO.java(164): "SELECT b.title FROM book_tab b
    WHERE b.code=@@null@@"
```

Fig. 4. A SQL query translated from the HQL query in Figure 3

Figure 2 shows a typical method that constructs an HQL query to list some books in a database, and Figure 3 shows the query string that we can extract from this method.

B. ORM-SQL Translation

The goal of this step is to handle data access points where we cannot extract a SQL query string, e.g. because an ORM library generates and sends the queries to the database server.

We implement this step by invoking the internal HQL to SQL compiler of Hibernate (`org.hibernate.hql.QueryTranslator`) for each HQL or JPA query with the same context that would be used for execution. Therefore, when data is accessed by HQL or JPA queries, we are able to generate the SQL query that would be sent to the database by Hibernate at runtime. A sample result of such a translation can be seen in Figure 4.

For data object accesses and criteria queries, however, we cannot use the internal translation engine of Hibernate. One possible solution here is to manually generate a SQL query based on the data objects accessed and the API methods used to access it. During this SQL generation, any uncertainty of dynamic constructs could be handled by using placeholders for unresolved query fragments as illustrated above. However, our current implementation does not support these data accesses.

C. SQL Parsing

Once we have all the potential data access points and all the native or translated SQL queries, the next step is to compare all these with the query that the developer is interested in. We perform this comparison at the level of *Abstract Syntax Trees* (ASTs) in order to have more flexibility for the comparison and to be able to handle unresolved query fragments. It requires a robust SQL parser which is able to handle the dialect of the targeted database management system, and to parse statements with unresolved query fragments.

Our current implementation has a SQL parser which supports the MySQL dialect and whenever it reaches an unresolved query fragment, it adds a special node to the AST that we call a *Joker node*. In Figure 5 we can see an AST constructed from the statement given in Figure 4.

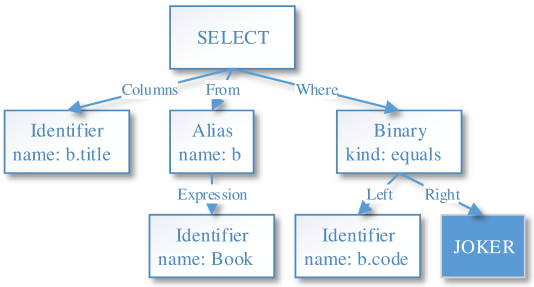

Fig. 5. Illustration of the AST of the query in Figure 4

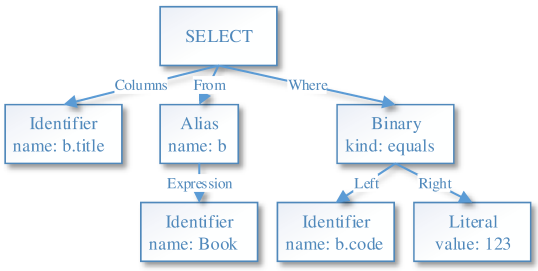

Fig. 6. Example AST matching the AST in Figure 5

D. Query Matching

The goal here is to find those queries that we extracted and have ASTs matching the AST of the query searched by the developer. This can be viewed as a clone matching technique where we attempt to find clones between the extracted and searched queries.

When we compare the ASTs, we follow our recursive definition of the matching relation $match_{exact}(t_i, t_j)$ between t_i and t_j trees, which we define as true if all the attributes of the root node of t_i ($root(t_i)$) (including the type of the node) are equal to the attributes of $root(t_j)$, and for all the t_{i_k} subtrees of $root(t_i)$ and t_{j_k} subtrees of $root(t_j)$, $match_{exact}(t_{i_k}, t_{j_k})$.

To handle the unresolved fragments, 'joker' nodes should match any other nodes or subtrees. That is, we define $match(t_i, t_j)$ as true if either t_i or t_j is a 'joker' node or $match_{exact}(t_i, t_j)$ is true.

Figure 6 shows a sample AST which is in $match$ relation with the AST in Figure 5. All the nodes are exactly the same, except the `Literal` node, which matches the 'joker' node in the tree.

III. Evaluation

We implemented our approach for systems written in Java and accessing a database via JDBC and/or Hibernate. To evaluate it, we tested the implementation on two open source systems, namely OSCAR EMR Clinical Management System[1] and OpenMRS Medical Record System[2]. Both systems use JDBC and Hibernate intensively and while OpenMRS uses 89 tables with a Java code base over 301 kLOC, Oscar has 480 tables and 2054 kLOC (see Table I). The size and the architecture of these systems allow us to demonstrate the efficiency of our approach in a real world environment.

[1]http://http://oscar-emr.com/

[2]http://openmrs.org/

A. Query Extraction and SQL Parsing

Table I shows the number of SQL queries that we successfully extracted and parsed from the source code of Oscar[3] and OpenMRS[4]. Both systems follow the Data Access Object pattern (but not strictly), hence most of the queries are prepared and sent to the database from DAO classes. It can be seen, however, that Oscar mixes the usage of JDBC and Hibernate, while OpenMRS uses Hibernate more extensively. The column of successfully parsed statements shows the number of queries for which we could successfully construct an AST. Queries that we cannot parse might contain syntactic errors or language constructs that our parser cannot handle in its current implementation state.

TABLE I
SIZE METRICS OF THE SYSTEMS AND THE NUMBER OF QUERIES EXTRACTED AND SUCCESSFULLY PARSED

System	LOC	Tables	JDBC Queries		Hibernate HQL/JPA	
			Extracted	Parsed	Extracted	Parsed
Oscar	2 054 940	480	123 643	123 298	27 242	9 005
OpenMRS	301 232	89	88	73	205	151

B. Query Matching

To play the role of the developer who seeks a problematic query, we collected SQL queries from execution traces of usage and testing scenarios. Both Oscar and OpenMRS have a test database available in their source repository, and their developers intensively use unit tests for basic functionalities and DAO implementations too. Oscar and OpenMRS have 1311 and 3258 test cases respectively, in their unit testing framework. We executed all these test cases and used log4jdbc[5] to trace database usage. For all the collected SQL queries, we saved the actual stack trace too. Then we filtered queries (based on their traces) that were sent to the database via JDBC or Hibernate HQL/JPA and we tried to locate them with our method.

The concept location task is injective: one query is sent to the database from exactly one location. However, one location can implement several queries. In fact, the same query string could be constructed in more locations too. Owing to this fact, and because of unresolved code fragments, we usually cannot report just the exact location where the query was sent to the database, but provide a set of matching locations. We treat (for the evaluation) this set as true positive if it contains the locations where the query was sent to the database, and false positive otherwise.

Table II and III show the number of queries and the true positive (TP) or false positive (FP) location sets, respectively, with the true or false positive ratios (TPR, FPR). Queries where the set of locations reported is empty are false negatives ($Queries - TP - FP$). Max, Avg, Var $Match$ stands for the maximum, average and variance values for the sizes of the sets of locations reported.

The results of JDBC reveal that we were able to identify most of the queries of Oscar usage scenarios with a TPR (which is actually equal to the recall in our case) of 87-99%.

[3]Checked out from GitHub at August 26, 2014.

[4]Checked out from GitHub at November 6, 2014.

[5]https://code.google.com/p/log4jdbc/

TABLE II
TRUE AND FALSE POSITIVE RATIO OF LOCATIONS REPORTED FOR JDBC QUERIES

System	Test Scenario	Queries	TP	TPR	FP	FPR	Max. Match	Avg. Match	Var.
Oscar	Billing	103	102	0.9903	0	0.0000	19	9.9706	8.4983
Oscar	Change Password	3	2	0.6667	0	0.0000	2	1.5000	0.5000
Oscar	First Login	3	2	0.6667	0	0.0000	2	1.5000	0.5000
Oscar	New Demographic	3	2	0.6667	0	0.0000	2	1.5000	0.5000
Oscar	Request Consultation	101	100	0.9901	0	0.0000	19	1.5000	10.1400
Oscar	Send Message	5	3	0.6000	0	0.0000	2	1.3333	0.4444
Oscar	Update User	3	2	0.6667	0	0.0000	2	1.5000	0.5000
Oscar	Writing Prescriptions	3	2	0.6667	0	0.0000	2	1.5000	0.5000
Oscar	Unit Tests	1005	650	0.6468	14	0.0139	4	1.5136	0.5027
OpenMRS	Unit Tests	39	34	0.8718	0	0.0000	4	2.1176	0.4948

TABLE III
TRUE AND FALSE POSITIVE RATIO OF LOCATIONS REPORTED FOR HIBERNATE QUERIES

System	Test Scenario	Queries	TP	TPR	FP	FPR	Max. Match	Avg. Match	Var.
Oscar	Add Provider	20	19	0.9500	0	0.0000	1	1.0000	0.0000
Oscar	Add Role	22	20	0.9091	0	0.0000	1	1.0000	0.0000
Oscar	Billing	702	294	0.4188	2	0.0028	5	1.0845	0.1598
Oscar	Change Password	39	33	0.8462	0	0.0000	1	1.0000	0.0000
Oscar	First Login	77	60	0.7792	0	0.0000	1	1.0000	0.0000
Oscar	New Demographic	67	47	0.7015	1	0.0149	5	1.1250	0.2344
Oscar	Request Consultation	498	150	0.3012	2	0.0040	5	1.0658	0.1264
Oscar	Send Message	43	36	0.8372	1	0.0233	2	1.0270	0.0526
Oscar	Update User	56	39	0.6964	0	0.0000	1	1.0000	0.0000
Oscar	Writing Prescriptions	100	67	0.6700	1	0.0100	2	1.0294	0.0571
Oscar	Unit Tests	1559	950	0.6094	23	0.0148	5	1.1079	0.1970
OpenMRS	Unit Tests	317	268	0.8454	0	0.0000	4	1.0672	0.1283

The results of Hibernate look promising too. Except for two scenarios, we were able to identify the origin of 60-95% of the queries by reporting almost everywhere just the matching location (see Avg. Match values). These results could probably be improved by getting a better parsed-extracted ratio for HQL/JPA queries in Oscar (see Table I).

Figure 7 shows a sample SQL query that we traced from the Billing scenario of Oscar and its origin, which is shown in Figure 8.

```
select billingser0_.billingservice_no as billings1_373_,
billingser0_.anaesthesia as anaesthe2_373_, billingser0_.
billingservice_date as billings3_373_, billingser0_.
description as descript4_373_, billingser0_.displaystyle as
displays5_373_, billingser0_.gstFlag as gstFlag373_,
billingser0_.percentage as percentage373_, billingser0_.
region as region373_, billingser0_.service_code as
service9_373_, billingser0_.service_compositecode as
service10_373_, billingser0_.sliFlag as sliFlag373_,
billingser0_.specialty as specialty373_, billingser0_.
termination_date as termina13_373_, billingser0_.value as
value373_ from billingservice billingser0_ where
billingser0_.service_code='A001A' and billingser0_.
billingservice_date=(select MAX(billingser1_.
billingservice_date) from billingservice billingser1_ where
billingser1_.billingservice_date<='2014-10-28' and
billingser1_.service_code='A001A');
```

Fig. 7. An example SQL query from the Billing scenario of Oscar

```
236 public Object[] getUnitPrice(String bcode, Date date) {
237   String sql = "select bs from BillingService bs where
    bs.serviceCode = ? and bs.billingserviceDate = ?";
238   Query query = entityManager.createQuery(sql);
239   query.setParameter(1,bcode);
240   query.setParameter(2, getLatestServiceDate(date,bcode)
    );
241
242   List<BillingService> results = query.getResultList();
243   ...
244 }
```

Fig. 8. The original HQL query and the Java code which prepares the query in Figure 7 (`BillingServiceDao.java`)

We collected the biggest number of distinct query strings from unit tests and got false positive reports because of methods in complex DAO classes where we could not extract query strings due to dynamic query construction.

C. Threats to Validity

Oscar and OpenMRS have been developed by two separate communities, hence, they have a different design and use a different coding style, e.g. for data accesses. It makes them good candidates for our technique, but so far our evaluation was performed only on them and it is possible that other systems might produce different results.

Our implementation is limited to some technologies as well. The query extraction technique is limited to JDBC and Hibernate and able to extract SQL, HQL and JPA queries and to identify data access points for criteria queries or simple Object accesses. The ORM-SQL translation is just able to handle HQL and JPA while the SQL parser supports only MySQL. Hence, we had to make the assumption for our evaluation that a developer seeks these kinds of queries and when we collected SQL strings, we filtered and kept only JDBC/Native/HQL/JPA queries. With this filtering, we also filtered statements that were also generated by Hibernate, e.g. because of two-pass (Eager/Lazy) fetching. We note, however, that for a general solution it would be necessary to support all the special features of the ORM library.

IV. RELATED WORK

Pioneer research on concept location dates back to the early '90s when Wilde et al. addressed the problem of *locating user functionalities* by "*using carefully designed test cases as probes to identify code that is closely related to a particular user functionality*" [6]. Since then, many researchers have studied different techniques, which they sometimes refer to as *feature/concern location/identification* too. Wilde et al. used a dynamic approach, but there are static techniques such as

information retrieval [2] or dependency analysis [7]; or it is also possible to combine static and dynamic approaches [8]. Dit et al. published a survey in which they identified 89 papers dealing with concept location [4].

One core step of our approach is the extraction of query strings from the source code, called the *query extraction*. Here, we rely on a technique by Meurice et al. [5] because of its ability to handle unresolved fragments. This area has also been intensively studied [9], [10] for different purposes such as quality assessment [11] and error checking [12], [13]. Some of these approaches are called as SQL fault localization [14] or fault diagnostics [15], but their goal is different (prevention by identifying erroneous SQL statements). Available techniques are usually unable to handle unresolved fragments of a query string, which allows us to compare extracted strings with concrete queries.

Our method for identifying data access points and extracting nested SQL queries has many other applications such as test generation [16], test coverage measurement [17], test selection [18], database semantic recovery [19], optimization [20] and impact analysis [21].

Compared to the above approaches, our study is, to the best of our knowledge the very first to propose a static analysis technique that addresses a specific concept location question for database applications: *where was this SQL query executed?*

V. Conclusions and Future Plans

There are several forum questions and blog posts like "*Finding the origin of a Query*"[6] and "*Backtrace from SQL query to application code*"[7] and because of the lack of static tools, they almost always recommend a dynamic solution. In contrast, as we pointed out earlier, dynamic analysis is not always feasible. Here, our goal was to devise a static analysis approach and to demonstrate its potential use.

Preliminary results show that a static technique can achieve good precision and recall with good true/false positive ratio in locating SQL queries sent to the database over JDBC or Hibernate. Although our implementation is just limited to these technologies, the main steps of the approach are general enough to be adapted to other technologies (e.g. a different ORM), and may possibly provide similar results.

As we mentioned above, we are not able to handle all the features of Hibernate, but we have preliminary solutions. Once we are able to handle Criteria queries and simple Object accesses, we can perform a thorough evaluation on more systems and usage scenarios.

Acknowledgements

The second author of this paper is supported by the F.R.S.-FNRS via the DISSE project.

References

[1] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," in *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*. IEEE Comp. Soc., 2005, pp. 33–42.

[2] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 23:1–23:34, 2012.

[3] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *Proceedings of the 18th International Conference on Program Comprehension (ICPC2010)*. IEEE, June 2010, pp. 14–23.

[4] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[5] L. Meurice, J. Bermudez, J. Weber, and A. Cleve, "Establishing referential integrity in legacy information systems - reality bites!" in *Proc. of 30th Interntional Conference on Software Maintenance and Evolution (ICSME2014)*. IEEE Comp. Soc., Oct. 2014.

[6] N. Wilde, J. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," in *Proceerdings of Conference on Software Maintenance, 1992*, Nov 1992, pp. 200–205.

[7] M. Trifu, "Improving the dataflow-based concern identification approach," in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR '09)*. IEEE Comp. Soc., 2009, pp. 109–118.

[8] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, Mar. 2003.

[9] B. Wiedermann, A. Ibrahim, and W. R. Cook, "Interprocedural query extraction for transparent persistence," *SIGPLAN Not.*, vol. 43, no. 10, pp. 19–36, Oct. 2008.

[10] M. Veanes, P. Grigorenko, P. Halleux, and N. Tillmann, "Symbolic query exploration," in *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM '09)*. Springer-Verlag, 2009, pp. 49–68.

[11] H. v. d. Brink, R. v. d. Leek, and J. Visser, "Quality assessment for embedded SQL," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '07)*. IEEE Comp. Soc., 2007, pp. 163–170.

[12] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007.

[13] M. Sonoda, T. Matsuda, D. Koizumi, and S. Hirasawa, "On automatic detection of SQL injection attacks by the feature extraction of the single character," in *Proceedings of the 4th International Conference on Security of Information and Networks (SIN '11)*. ACM, 2011, pp. 81–86.

[14] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL faults in database applications," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Comp. Soc., 2011, pp. 213–222.

[15] M. A. Javid and S. M. Embury, "Diagnosing faults in embedded queries in database applications," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops (EDBT-ICDT '12)*. ACM, 2012, pp. 239–244.

[16] K. Pan, X. Wu, and T. Xie, "Guided test generation for database applications via synthesized database interactions," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, pp. 12:1–12:27, Apr. 2014.

[17] M. J. Suárez-Cabal and J. Tuya, "Using an SQL coverage measurement for testing database applications," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 253–262, Oct. 2004.

[18] D. Willmor and S. M. Embury, "A safe regression test selection technique for database-driven applications," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Comp. Soc., 2005, pp. 421–430.

[19] A. Cleve, J.-R. Meurisse, and J.-L. Hainaut, "Database semantics recovery through analysis of dynamic SQL statements," in *Journal on Data Semantics XV*, S. Spaccapietra, Ed. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Database Semantics Recovery Through Analysis of Dynamic SQL Statements, pp. 130–157.

[20] W. Kim, "On optimizing an SQL-like nested query," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, Sep. 1982.

[21] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, 2008, pp. 451–460.

[6] http://java.dzone.com/articles/hibernate-debugging-where-does

[7] http://stackoverflow.com/questions/12631315/backtrace-from-sql-query-to-application-code