Static Analysis of Dynamic Database Usage in Java Systems

Loup Meurice, Csaba Nagy, Anthony Cleve*

PReCISE Research Center, University of Namur, Belgium

Abstract. Understanding the links between application programs and their database is useful in various contexts such as migrating information systems towards a new database platform, evolving the database schema, or assessing the overall system quality. In the case of Java systems, identifying which portion of the source code accesses which portion of the database may prove challenging. Indeed, Java programs typically access their database in a dynamic way. The queries they send to the database server are built at runtime, through String concatenations, or Object-Relational Mapping frameworks like Hibernate and JPA. This paper presents a static analysis approach to program-database links recovery, specifically designed for Java systems. The approach allows developers to automatically identify the source code locations accessing given database tables and columns. It focuses on the combined analysis of JDBC, Hibernate and JPA invocations. We report on the use of our approach to analyse three real-life Java systems.

Keywords— Database access recovery, static analysis, Java, ORM

1 Introduction

In various maintenance and evolution scenarios, developers have to determine which portion of the source code of their applications accesses (a given fragment of) the database. Let us consider, among others, the cases of database reverse engineering, database refactoring, database platform migration, service identification, quality assessment or impact analysis for database schema change. In the context of each of these processes, one needs to identify and analyze all the database queries executed by the application programs.

In the case of systems written in Java, the most popular programming language today [1], database manipulation has become increasingly complex in recent years. Indeed, a large-scale empirical study, carried out by Goeminne et al. [8], reveals that a wide range of dynamic database access technologies are used by Java systems to manipulate their database. Those access mechanisms partly or fully hide the actual SQL queries executed by the programs [6]. Those queries are *generated* at run time before they are sent to the database server.

In this paper, we address this problem of recovering the traceability links between Java programs and their database in presence of such a level of dynamicity. We propose a static analysis approach allowing developers to identify the

* This research is supported by the F.R.S.-FNRS via the DISSE project.

source code locations where database queries are executed, and to extract the set of actual SQL queries that could be executed at each location. The approach is based on algorithms that operate on the call graph of the application and the intra-procedural control-flow of the methods. It considers three of the most popular database access technologies used in Java systems, according to [8], namely JDBC, Hibernate, and JPA. We evaluated our approach based on three real-life open-source systems with size ranging from 250 to 2,054 kLOC accessing 88 – 480 tables in the database. We could extract queries for 71.5% – 99% of database accesses with 87.9% – 100% of valid queries.

The paper is organized as follows. Section 2 introduces the three database access technologies considered by our approach. Section 3 presents our approach and illustrates it through examples. Section 4 reports on the use of our approach to analyze real-life Java systems. A related work discussion is provided in Section 5. Concluding remarks are given in Section 6.

2 Java Database Access Technologies

Below we briefly introduce JDBC, Hibernate and JPA, by illustrating their underlying database access mechanisms.

JDBC The JDBC API is the industry standard for database-independent connectivity between the Java programming language and relational databases. It provides a call-level API for SQL-based database access, and offers the developer a set of methods for querying the database, for instance, methods from `Statement` and `PreparedStatement` classes (see Figure 1).

```
1  public class ProviderMgr {
2     private Statement st;
3     private ResultSet rs;
4     private boolean ordering;
5
6     public void executeQuery(String x, String y){
7        String sql = getQueryStr(x);
8        if(ordering)
9           sql += " order by " + y;
10       rs = st.execute(sql);
11    }
12    public String getQueryStr(String str){
13       return "select * from " + str;
14    }
15    public Provider[] getAllProviders(){
16       String tableName = "Provider";
17       String columnName = (...) ? "provider_id" : "provider_name";
18       executeQuery(tableName, columnName);
19       ...
20    }}
```

Fig. 1. Java code fragment using the JDBC API to execute a SQL query (line 10).

Hibernate Hibernate is an Object-Relational Mapping (ORM) library for Java, providing a framework for mapping an object-oriented domain model to a traditional relational database. Its primary feature is to map Java classes to database tables (and Java data types to SQL data types). Hibernate provides also an

SQL-inspired language called *Hibernate Query Language* (HQL) which allows to write SQL-like queries using the mappings defined before. Figure 2 (1) provides an example of an HQL query execution (line 13). In addition, *Criteria Queries* are provided as an object-oriented alternative to HQL, where one can construct a query by simple method invocations. See Figure 2 (2) for a sample usage of a Criteria Query. Hibernate also provides a way to perform *CRUD operations* (Create, Read, Update, and Delete) on the instances of the mapped entity classes. Figure 2 (3) illustrates a sample record insertion in the database.

(1) Java code executing a HQL query (line 13). Product selection according to its category

```
1  public class ProductDaoImpl implements ProductDao {
2
3      private SessionFactory sessionFactory;
4
5      public void setSessionFactory(SessionFactory sessionFactory) {
6          this.sessionFactory = sessionFactory;
7      }
8
9      public Collection loadProductsByCategory(String category) {
10         return this.sessionFactory.getCurrentSession()
11                 .createQuery("from Product product where category=?")
12                 .setParameter(0, category)
13                 .list();
14     }}
```

(2) Java code executing a Criteria query. Customer selection restricted on the name and city

```
1  List cats = sess.createCriteria(Customer.class)
2      .add( Restrictions.eq("name", "Smith") )
3      .add( Restrictions.in( "city", new String[] { "New York", "Houston
           ", "Washington DC" } )
4      .list();
```

(3) Hibernate operation on a mapped entity class instance. Insertion of a new customer

```
1  private static Session session;
2  ...
3
4  public static void saveCustomer(Customer myCustomer){
5     saveObject(myCustomer));
6  }
7
8  public static void saveObject(Object o){
9     session.save(o);}
```

Fig. 2. Samples of Hibernate accesses.

Java Persistence API JPA is a Java API specification to describe the management of relational data in applications. Just like Hibernate, JPA also provides a higher level of abstraction based on the mapping between Java classes and database tables permitting operations on objects, attributes and relationships instead of tables and columns. It offers the developers several ways to access the database. One of them is the *Java Persistence Query Language* (JPQL), a platform-independent object-oriented query language which is defined as part of the JPA API specification. JPQL is used to make queries against entities stored in a relational database. Like HQL, it is inspired by SQL, but it operates on JPA entity objects rather than on database tables. Figure 3 (1) shows an example of JPQL query execution. JPA also provides a way to perform CRUD operations

(1) Sample JPQL query. Customer selection according to a given id

```
1 EntityManagerFactory emf = ...;
2 EntityManager em = emf.createEntityManager();
3 Order order = ...;
4 Integer cust_id = order.getCustomerId();
5 Customer cust = (Customer)em.createQuery("SELECT c FROM Customer c
      WHERE c.cust_id=:cust_id")
6     .setParameter("cust_id", cust_id).getSingleResult();
```

(2) JPA operation on a mapped entity class instance. Creation and insertion of a new order

```
1 EntityManager entityManager = entityManagerFactory.createEntityManager
      ();
2 entityManager.getTransaction().begin();
3 Order order= createNewOrder();
4 entityManager.persist( order );
5 entityManager.getTransaction().commit();
6 entityManager.close();
```

Fig. 3. Samples of JPA accesses.

on the instances of mapped entity classes. For instance, Figure 3 (2) illustrates the creation and insertion of a new order in the database.

3 Approach

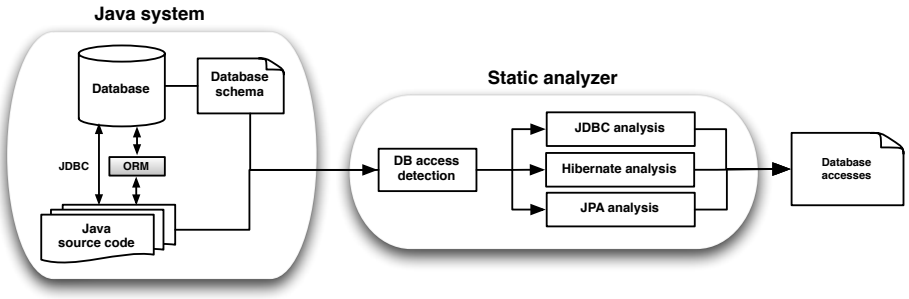

Fig. 4. Overview of the proposed approach.

Figure 4 presents an overview of our approach which combines three different analyses: the JDBC, Hibernate and JPA analyses. The output of the full process is a set of database access locations and the database objects (tables and columns) impacted/accessed by a given access. Those database objects are detected based on the actual database schema.

3.1 Initial Analysis

Call Graph Extraction The complete recovery of a query executed in a given Java method is a complex process. In most cases, a SQL query (a database access in general) is constructed using some of the input parameters of the given method. For instance, the `executeQuery` method in Figure 1 uses its parameters for constructing the SQL query. Consequently, the local recovery of the query is not sufficient and the exploration of the call graph of that given method is

necessary for determining the different possible values of the needed parameters. We designed an approach based on inter-procedural analysis in order to deal with the call graph reconstruction and the extraction of every possible value of the parameters used in the query construction.

Database Access Detection The database access detection step aims to detect all the source code locations querying the database by means of a JDBC/Hibernate/JPA method. Our Java analyzer constructs an abstract syntax tree and uses a visitor to navigate through the different Java nodes and expressions. We defined an exhaustive list of JDBC, Hibernate and JPA methods accessing the database (based on the documentation of each technology). Our detection is designed to detect the calls of those methods and to send them to the corresponding analysis (JDBC, Hibernate or JPA analysis).

3.2 JDBC Analysis

Our JDBC analysis focuses on the database accesses using the JDBC API, where we follow a two-phase process. As illustrated in Section 2, a JDBC access recovery can be seen as a String expression recovery. Once our analyzer has detected a JDBC access, it will then recover the corresponding SQL query. Finally, our SQL parser constructs the abstract syntax tree of the SQL query and identifies which part of the database schema is involved in that access; that is, the parser identifies the database tables and columns accessed with it. This identification relies on the actual database schema.

Algorithm 1 formalizes the first phase allowing the recovery of all the possible string values of an expression (a more detailed description of the used procedures is given in Algorithm 2). First, we *locally* resolve the expression and then we deal with the call graph extraction, when it is necessary. Let us apply Algorithm 1 on the sample code in Figure 1. This algorithm gets executed when the *Database Access Detection* finds a JDBC-based data access, i.e., `st.execute(sql)`. Here, `sql` is the String expression which will be recovered by the algorithm and which is located in the method `executeQuery(String x, String y)`. These two elements will be the inputs of the algorithm. First, the algorithm extracts the possible local values of `sql`, i.e., '`select * from` x' and '`select * from` x `order by` y' (line 2). Then it deals with the x and y input parameters by extracting the call graph first. Analyzing the call graph allows us to recover the possible values of the parameters. We illustrate this step for each possible value.

Let $value =$ '`select * from` x'; x is the only parameter of the `executeQuery` method (line 4). The algorithm explores the code for retrieving the expressions invoking the `executeQuery` method (line 8). It returns only one call expression, namely `executeQuery(tableName, columnName)`. The next step is to retrieve $tableName$ (line 10), the input expression corresponding to x. For this step, we recursively resolve the expression $tableName$ (line 13); the result is '`Provider`'. Then, we replace all the input parameters with their corresponding values obtained earlier (line 15). In this example, we merely replace x with '`Provider`' and thus, the resulting value for the query string is '`select * from Provider`'.

```
Procedure recoverExpr(Expression expr, Method method)
Input: a Java expression representing a String value and the Java method where the
       expression is located.
Output: the list of every possible String values corresponding to this expression.
1    Expr[] result = initialize()
     // Locally extracting the values of the given expression
2    Expr[] values = getLocalValues(expr, method)
3    for value ∈ values do
         //Extracting the used input parameters from the current value
4        Variable[] inputs = getInputParams(method, value)
5        if inputs = null then
6            result.add(value)
7        else
             //Extracting the call graph of the given method in order to recover
             the value of each used input
8            MethodCallExpr[] callGraph = callGraph(method)
9            for call ∈ callGraph do
                 //Extracting the input values from the current call
10               Expr[] inputExprs = extractParamValues(method, call, inputs)
11               Expr[][] inputValues = initialize()
12               for inputExpr ∈ inputExprs do
                     //Recursive call for each input
13                   inputValues.add(recoverExpr(inputExpr, inputExpr.method()))
14               end
                 //Replacing each input by the obtained values
15               Expr[] product = replaceInput(value, inputs, inputValues)
16               for e ∈ product do
17                   result.add(e)
18               end
19           end
20       end
21   end
22   return result
```

Algorithm 1: Algorithm for recovering the string values of a given Java expression.

Let *value* = '`select * from` x `order by` y'; the process is slightly different. In this case there are two input parameters: x and y. The result for x is the same as above ('`Provider`'), but y, reduced to *columnName*, may correspond to two different values: '`provider_id`' and '`provider_name`'. The algorithm returns two possible values (line 15): '`select * from Provider order by provider_id`' and '`select * from Provider order by provider_name`'.

The final result of the algorithm will be 3 different string values for the *sql* expression: '`select * from Provider`', '`select * from Provider order by provider_id`', and '`select * from Provider order by provider_name`'. In the end of the process, the SQL parsing phase will point to the `Provider` table and its `provider_id` and `provider_name` columns as the accessed objects.

3.3 Hibernate Analysis

Similarly to the JDBC API, Hibernate provides the developer multiple database access/query mechanisms. The aim of the Hibernate analysis is to identify the source code locations accessing the database through Hibernate. While it partly relies on the JDBC analysis and its algorithm of string value recovery, the Hibernate analysis is more sophisticated due to the ORM complexity.

Like the JDBC API, Hibernate also proposes different Java methods to execute either native SQL queries or HQL queries. The extraction process of those

Procedure `getLocalValues`(*Expr expr, Method method*)
Input: A Java expression representing a String value and the Java method where the expression is located
Output: All the possible values of the given expression by only exploring the given local method.
Procedure `getInputParams`(*Method method, Expr expr*)
Input: A Java method declaration and a Java expression.
Output: The input parameters of the given method which are part of the given expression
Example:
- method = public static void printCustomer(Connection con, Integer id)
- expr = "select * from Customer where $cust\_id$ = " + id
- res = [id]

Procedure `callGraph`(*Method method*)
Input: A Java method declaration.
Output: The Java expressions invoking the given method.
Procedure `extractParamValues`(*Method method, MethodCallExpr mce, Variable[] inputs*)
Input: A Java method declaration, a Java expression invoking the given method and a set of input parameters of the given method.
Output: The corresponding value of each parameter.
Example:
- method = public static void printCustomer(Connection con, Integer id)
- mce = printCustomer(myConnection, 201456)
- inputs = [id]
- res = [201456]

Procedure `replaceInput`(*Expr expr, Variable[] inputs, Expr[][] inputValues*)
Input: A Java expression, a list of variables used by the given expression, the possible values of each variable
Output: Replacing the variables part of the given expression by their corresponding values
Example:
- expr = "select * from Customer where $first\_name$ = " + firstName + " and $last\_name$ = " + lastName
- inputs = [firstName, lastName]
- inputValues = [['James', 'John'], ['Smith']]
- res = [select * from Customer where $first\_name$ = 'James' and $last\_name$ = 'Smith', select * from Customer where $first\_name$ = 'John' and $last\_name$ = 'Smith']

Algorithm 2: Description of the procedures used in Algorithm 1

queries is similar to the JDBC analysis process (Algorithm 1). However, our HQL parser is slightly different from the parser of the JDBC analysis. Indeed, at this point we cannot just extract a SQL query string. Thus, we implemented a feature to be able to translate an HQL query into the corresponding SQL query. This translation is processed by invoking the internal HQL to SQL compiler of Hibernate (`org.hibernate.hql.QueryTranslator`) with the same context that would be used for execution. Once we obtained the corresponding translated SQL query, we are able to parse it and extract the involved objects.

Furthermore, as previously described, Hibernate also offers a set of methods operating on instances of mapped entity classes, e.g., Figure 2 (3). This way of accessing the database cannot be reduced to a mere string recovery process. Instead, the purpose is to determine the Java class of the object. The proposed solution consists in firstly determining the entity class(es) of the input object and then, detecting the corresponding mapped database objects. This last phase analyzes the Hibernate mapping files of the system. These mapping files instruct Hibernate how to map the defined class or classes to the database tables. We did not present our algorithm allowing to determine the entity class of an input Java object because it uses the same logic (but simplified) that Algorithm 1. Instead, we illustrate the use of that algorithm on Figure 2 (3). The Database Access Detection detects `session.save(o)` as a database access. *o* is the expression to resolve and it is located in `saveObject(Object o)`. *o* is identified as an input

parameter of the method *saveObject*. Then, the algorithm explores the code to retrieve the expressions invoking the *saveObject* method (call graph extraction). Only one call expression is returned, namely `saveObject(myCustomer)`. Next, we recursively resolve the *myCustomer* expression. *myCustomer* is also a parameter of the `saveCustomer` method, however, there is no call expression for it (empty call graph). Thus, we resolve *myCustomer* locally: by exploring the `saveCustomer` method, we detect that *myCustomer* is an instance of the `Customer` class. This step will, therefore, return the `Customer` class as the only solution for the *o* expression. Finally, our solver will detect the mapping between the `Customer` class and its corresponding database table.

3.4 JPA Analysis

The JPA analysis concentrates on the database accesses by means of JPA. Like Hibernate, JPA proposes Java methods to execute either native SQL queries or JPQL queries. The extraction process of those queries is similar to the Hibernate analysis: we rebuild the query value by means of Algorithm 1 and then we parse the JPQL query. The JPQL parser uses the same approach as for HQL, by invoking the internal HQL to SQL compiler of Hibernate.

Like Hibernate, JPA also permits accessing the database by operating on Java instances of mapped entity classes, e.g., Figure 3 (2). We use the same approach to address that problem. However, instead of using the Hibernate mapping files for establishing the mapping between the entity classes and the database tables, the DB Mapper will consider the JPA annotations which define this mapping.

3.5 Process Output

The output of the full process is the set of the database accesses detected by our static analysis as well as the code location of each access and the database tables and columns involved in it. The code location of a given access is expressed by the minimal *program path* necessary for creating and executing the database access. The below example shows sample information gathered for a database access where a SQL query is executed at line 124 in `DatabaseUtil.java`. The current method in which the query execution occurs is called by `CheckDrug OrderUnit.java` at line 56. The database objects involved in this query are the *drug_order* table and *units*, one of its columns.

JDBC access: `'SELECT DISTINCT units FROM drug_order WHERE units is NOT NULL'`
Program path: [**CheckDrugOrderUnit.java**, line=**56**] → [**DatabaseUtil.java**, line=**124**]
Database schema objects:
↪ Database Tables: [**drug_order**]
↪ Database Columns: [**drug_order.units**]

4 Evaluation

In this section we evaluate our approach on three real-life systems. The detailed results of this evaluation are available as an online appendix[1].

[1] `https://staff.info.unamur.be/lme/CAISE16/`

4.1 Evaluation Environment

Table 1 presents an overview of the main characteristics of the target systems. Oscar (`oscar-emr.com`) is an open-source information system that is widely used in the healthcare industry in Canada. The source code comprises approximately two million lines of code. OSCAR combines JDBC, Hibernate and JPA to access the database. OpenMRS (`openmrs.org`) is a collaborative open-source project to develop software to support the delivery of health care in developing countries (mainly in Africa). OpenMRS uses a MySQL database accessed via Hibernate and dynamic SQL (JDBC). Broadleaf Commerce (`broadleafcommerce.org`) is an open-source, e-commerce framework written entirely in Java on top of the Spring framework. Broadleaf uses a relational database accessed via JPA.

Table 2 contains the results of the process of identifying database accesses applied to the three systems. For each system and technology supported, it presents the total number of locations accessing the database.

Table 1. Size metrics of the systems

System	Description	LOC	Tables	Columns
Oscar	Medical record system	2 054 940	480	13 822
OpenMRS	Medical record system	301 232	88	951
Broadleaf	E-commerce framework	254 027	179	965

Table 2. Number of database access locations per technology

System	Database Accesses		
	JDBC	Hib	JPA
Oscar	123 661	727	31 729
OpenMRS	77	687	0
Broadleaf	0	0	930

Table 3. Complexity of database access recovery

	JDBC		Hib		JPA	
	$\bar{x}$	max	$\bar{x}$	max	$\bar{x}$	max
Oscar	4	8	1.5	3	3.8	7
OpenMRS	1.2	3	1	2	-	-
Broadleaf	-	-	-	-	1	1

Figure 5 shows the set of tables and columns accessible by the different technologies. In the Oscar system, we notice that JDBC remains the most widely used technology regarding the number of different columns accessed (10,350 columns accessed from 123,661 source code locations). Concerning OpenMRS, the biggest database part is accessed by Hibernate (713 columns for 687 locations) whereas JPA is the only used mechanism in Broadleaf (431 columns for 930 locations).

Table 3 depicts, for each system, the algorithmic complexity in terms of the number of recursive calls needed for completely recovering a code location accessing the database, i.e., the number of recursive calls in Algorithm 1. In Oscar, one can notice that 4 recursive calls are required, on average, to fully reconstruct a database access via JDBC, while the *most complex* detected accesses require 8 recursive calls. By comparing with the other systems in Table 3, we note that Oscar is the most complex, recursive calls being often necessary to recover the database accesses. In contrast, we can observe that in OpenMRS and Broadleaf, most database accesses are built within the same method.

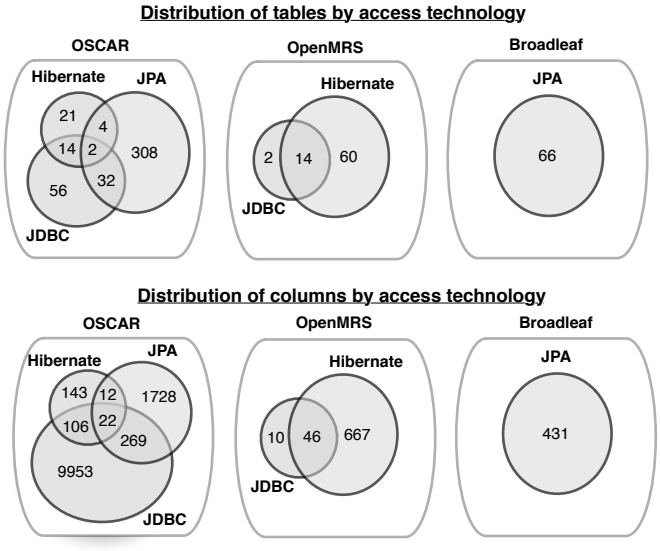

Fig. 5. Distribution of *tables and columns* by access technology

4.2 Successfully extracted queries

The oracle To evaluate the effectiveness of our approach in extracting database accesses, we assess whether we can identify most of the database accesses and also the noise of the technique. First we need to have a ground truth, i.e., the *actual* set of queries that are sent to the database with their corresponding source code locations. Once we have this set of queries, we can compare them to our extracted set of queries. However, the availability of a complete ground truth is not a realistic working assumption in the context of large legacy systems.

The *oracle* that we used for assessing our approach is the set of unit tests of each software system. That is, we systematically collected all the database accesses (JDBC, Hibernate and JPA) produced by the execution of the test suites. We gathered this query set by analyzing trace logs of the execution of the unit tests of each system. To do so, we used our modified version of *log4jdbc*[2] to collect trace logs containing the exact string values of all of the queries sent to the database and their corresponding stack traces.

Table 4. Coverage values of the unit tests

System	Test LOC	Test Runs	Covered Classes HIB/JPA	Covered Classes JDBC	Covered Locations HIB/JPA	Covered Locations JDBC
Oscar	49 086	1 311	65.00%	1.05%	56.79%	0.28%
OpenMRS	76 960	3 258	69.57%	16.00%	58.16%	13.56%
Broadleaf	17 633	255	33.96%	-	19.10%	-

[2] `http://code.google.com/p/log4jdbc/`

Table 4 presents statistics about the unit tests of the systems under question. We count the number of test runs reported by the build system and show the total lines of Java code in the testing directories. In addition to the test classes, there are functional tests (e.g., Broadleaf uses Groovy tests), resulting that the number of executed test cases are larger than the number of test classes. In general, all the systems are well tested with unit tests, and developers do not just test core functionality of their systems, but the testing of DAO classes is also one of their main goals. All the systems have test databases and hundreds of test cases for testing database usage. Thus, it is reasonable to consider as oracle the data accesses collected through the execution of the unit tests.

The queries that we identify with the help of *log4jdbc* are filtered based on their stack traces, in order to distinguish between queries sent to the database directly through JDBC or Hibernate. Also, this filtering keeps queries generated by Hibernate explicitly for HQL or JPA queries and filters those implicit queries, which are generated for caching or lazy data fetching purposes, for instance.

Table 4 shows what percentages of the classes and locations (that we extracted as data accesses) are covered by the unit tests. For Oscar and OpenMRS, the two largest systems that we analyzed, this coverage value is 65% for the classes where we found Hibernate or JPA queries. The coverage value of JDBC data accesses is, however, quite low for all systems. The reason for this is that these systems implement main features using ORM technologies, and it mostly happens out of the scope of the main features where they use JDBC to accesses the database, e.g., in utility classes for upgrading the database, or in classes to prepare test databases. These parts of the code are usually not tested by unit tests, resulting in low coverage for our analysis.

Percentages of successfully extracted queries Conceptually, the number of possible queries is infinite (i.e., when a part of a query depends on user input, its value could be anything). However, to assess if we were able to identify most of the database accesses or not, we calculate the percentages of *successfully extracted* and *unextracted* queries. We consider a query of the oracle (a query logged in the execution traces of the unit tests) *successfully extracted* if we could also extract it from the source code. Otherwise, we consider it *unextracted*. In other words, successfully extracted queries are the true positive queries, while the unextracted ones are the false negatives. To determine if a query in the oracle was successfully extracted or not, we compare the stack trace of all of these queries to the 'program paths' (see Section 3.5) of the extracted queries. Moreover, we compare the string values of the SQL queries.

Table 5. Percentage of successfully extracted queries for each system

System	Technologies		Total
	JDBC	HIB/JPA	
Oscar	1681/2038	892/1558	71,5%
OpenMRS	31/41	268/322	82,4%
Broadleaf	-	94/95	99%

Table 6. Percentage of valid queries for each system

System	Technologies		Total
	JDBC	Hib/JPA	
Oscar	14/17	656/689	94.9%
OpenMRS	8/8	86/99	87.9%
Broadleaf	-	29/29	100%

Table 5 shows the percentages of the successfully extracted queries. For assessing the JDBC analysis on Oscar, we found 2,038 queries in the trace logs, among which our approach successfully extracts 1,681. Regarding the Hibernate/JPA analysis, we identified 892 queries out of 1,558. In general, we identified 71.5% of the queries. In the case of OpenMRS, for the JDBC analysis, we identified 31 queries out of 41, while we identified 268 Hibernate/JPA accesses out of 322. In total, we identify 82.4% of the queries. For Broadleaf, the percentage of successfully extracted queries is 99% (94 JPA accesses out of 95).

Percentage of valid queries It is possible that we extract a query, and we report it as valid, but it is never constructed in the code. Hence it is *invalid*. It can happen when our static technique fails to deal with constructs in the code which would require additional information that we cannot extract statically, e.g., evaluating conditional statements (see Section 4.3). We consider these queries as the noise of our approach. In other words, these queries are the false positive queries reported by our technique.

We limit the assessment to those database access points that are covered by the unit tests. If the tests cover an access point, we can make the assumption that the possibly valid queries on that location were sent to the database and traced by our dynamic analysis. All the queries that were reported for these locations, and are not in the oracle, are thus considered as invalid (false positives).

Results are presented in Table 6. In the case of Oscar, with the JDBC analysis we obtain 14 valid out of 17 queries and 656 valid out of 689 for the Hibernate/JPA analysis. The percentage of the valid queries value is 94.9%. For OpenMRS, we obtain a percentage of 87.9% with 8 true positive out of 8 for the JDBC analysis and 86 true positives out of 99 for the Hibernate/JPA analysis. Finally, for Broadleaf there are no invalid queries (29 true positives out of 29).

4.3 Limitations

As we have seen, our approach reached good results when applied to real-life Java systems. However, we identified some limitations of our approach that are mainly due to its static nature. Below, we give an overview of those limitations that may cause failures in the automated extraction of (valid) SQL queries.

String manipulation classes The standard Java API provides developers with classes to manipulate String objects, such as StringBuilder and StringBuffer. The main operations of those classes are the *append* and *insert* methods, which are overloaded so as to accept data of any type. In particular, a StringBuilder/StringBuffer may be used for creating a database access (e.g., a SQL query). The current version of our analysis does not handle the use of those classes in the string value recovery process. This is one reason for some unsuccessfully extracted queries. As we manually investigated it for the OpenMRS system, among the 54 Hibernate/JPA accesses not extracted by our parser (see Table 5), 49 are due to the use of StringBuilder objects for creating the query value. This obviously affects the percentage of successfully extracted queries[3].

[3] Example of the use of *StringBuilder* to create a SQL query: `http://bit.ly/1XNeL4e`

User-given inputs Similarly, executed SQL queries sometimes include input values given by the application users. This is the case in highly dynamic applications that allow users to query the database by selecting columns and/or tables in the user interface. In such a situation, which we did not encounter in our evaluation environment, our approach can still detect the database access location but the static recovery of the associated SQL queries may be incomplete.

Boolean conditions Another limitation we observed relates to the conditions in *if-then*, *while*, *for*, and *case* statements. Our parser is designed to rebuild all the possible string values for the SQL query. Thus, it considers all the possible program paths. Since our *static* analyzer is unable to resolve a boolean condition (a *dynamic* analysis would be preferable), these cases generate some noise (false positive queries). In the three subjects systems, a total of 12 invalid queries were extracted by our approach due to boolean conditions[4].

5 Related Work

The key novelty of our approach relies on the static reconstruction of SQL queries from Java source code in the presence of Object-Relational Mapping frameworks such as Hibernate and JPA. In particular, we are not aware of another approach supporting such a task in the case of *hybrid* database access mechanisms, where JDBC, Hibernate, and JPA accesses *co-exist* in the same information system.

Several previous papers identify database accesses by extracting dynamically constructed SQL queries (e.g., for JDBC-based database accesses). The purpose of these approaches ranges from error checking [4, 9, 15, 17], SQL fault localization [5], fault diagnosis [10] to impact analysis for database schema changes [11, 16]. A pioneer work was published by Christensen *et al.* [4], who propose a static string analysis technique that translates a given Java program into a flow graph, and then analyzes the flow graph to generate a finite-state automaton. They evaluate their approach on Java classes with at most 4 kLOC. Gould *et al.* propose a technique close to a pointer analysis, based on an interprocedural data-flow analysis [9,17]. Maule *et al.* use a similar k-CFA algorithm and a software dependence graph to identify the impact of relational database schema changes upon object-oriented applications [11]. van den Brink *et al.* present a quality assessment approach for SQL statements embedded in PL/SQL, COBOL and Visual Basic code [2]. The initial phase of their method consists in extracting the SQL statements from the source code using control and data-flow analysis techniques. They evaluate their method on COBOL programs with at most 4 kLOC. Ngo and Tan [14] make use of symbolic execution to extract database interaction points from web applications. Through a case study of PHP applications with sizes ranging 2 – 584 kLOC, they show that their method is able to extract about 80% of such interactions.

[4] Example of invalid extracted query: *"from Concept as concept left join concept.names as names where names.conceptNameType = 'FULLY_SPECIFIED' and concept.retired = false order by concept.conceptId asc"*. `http://bit.ly/1YOTJAT`

Compared to the above previous approaches [2, 4, 9, 11, 14, 17], our SQL extraction technique does not require an expensive data-flow analysis nor symbolic execution. Its input is the abstract syntax tree, and it relies on the intraprocedural control flow of the methods associated with their call graph. This makes the approach applicable to large-scale Java applications, as shown in this paper. In addition, the above approaches are not directly applicable to ORM-based Java systems.

There are only a few studies targeting applications using ORM frameworks, particularly Java applications using Hibernate. Goeminne *et al.* [7] study the co-evolution between code-related and database-related activities in data-intensive systems combining several ways to access the database (native SQL queries and Object-Relational Mapping). Their analysis remains at the granularity level of source code files, and does not involve the fine-grained inspection of the ORM queries. Chen *et al.* [3] propose an automated framework for detecting, flagging and prioritizing database-related performance anti-patterns in applications that use object-relational mapping. In this context, the authors identify database-accessing code paths through control-flow and data-flow analysis, but they do not reconstruct statically the SQL queries that correspond to the identified ORM code fragments. Instead, they execute the applications and rely on *log4jdbc* to log the SQL queries that are executed. The above papers study the peculiarities of ORM code, but they do not contribute to database usage analysis in general, nor to query reconstruction in particular. Our approach is, therefore, the first static analysis technique able to identify database accesses in Java systems that rely on an ORM framework and to translate them to queries sent to the database.

In our recent work, we applied an *earlier* version of our approach to two usage scenarios. First, we were able to elicit implicit foreign keys in a Java system [12], based on the analysis of JDBC invocations. We analyzed both the database schema and the schema recovered from the source code, but the Hibernate and JPA analysis was only limited to the analysis of the schema mapping files and annotations, used as heuristics. Second, in [13], we conducted a study on locating the source code origin of a SQL query executed on the database side. While this short paper relies on our query extraction approach, it focuses on the algorithm for matching *one* concrete SQL query against others. In this paper, we significantly extend our query extraction technique towards a *hybrid* approach by complementing it with the Hibernate and JPA analyses, and we perform an experimental evaluation of its accuracy based on real-life information systems.

6 Conclusions and Future directions

We presented a static analysis approach that allows developers to identify and analyze database access locations from highly dynamic Java systems. Our approach is able to handle Java systems that combine JDBC-based data accesses with the usage of Hibernate and/or JPA as popular object-relational mapping technologies. The evaluation shows that the proposed approach can successfully extract queries for 71.5% – 99% of database accesses with 87.9% – 100% of valid

queries. Although we identified some limitations (as we presented above), we argue that our approach is applicable in practice to real-life Java projects, and can achieve useful results for further analyzes.

In our future work, we plan to extend our results to other programming languages and database platforms. We also intend to empirically analyse database usage evolution practices, and to study program-database co-evolution patterns. Our ultimate goal is to support developers in the context of software evolution scenarios such as database schema change and database platform migration.

References

1. Tiobe programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, accessed: 2016-02-01
2. Brink, H.v.d., Leek, R.v.d., Visser, J.: Quality assessment for embedded SQL. In: SCAM '07. pp. 163–170. IEEE Comp. Soc. (2007)
3. Chen, T.H., Shang, W., Jiang, Z.M., Hassan, A.E., Nasser, M., Flora, P.: Detecting performance anti-patterns for applications developed using object-relational mapping. In: ICSE '14. pp. 1001–1012. ACM (2014)
4. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: SAS'03. pp. 1–18. Springer-Verlag (2003)
5. Clark, S.R., Cobb, J., Kapfhammer, G.M., Jones, J.A., Harrold, M.J.: Localizing SQL faults in database applications. In: ASE '11. p. 213. IEEE (2011)
6. Cleve, A., Mens, T., Hainaut, J.L.: Data-intensive system evolution. IEEE Computer 43(8), 110–112 (August 2010)
7. Goeminne, M., Decan, A., Mens, T.: Co-evolving code-related and database-related changes in a data-intensive software system. In: CSMR-WCRE '14. pp. 353–357 (2014)
8. Goeminne, M., Mens, T.: Towards a survival analysis of database framework usage in Java projects. In: ICSME'15 (2015)
9. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: ICSE '04. pp. 645–654. IEEE (2004)
10. Javid, M.A., Embury, S.M.: Diagnosing faults in embedded queries in database applications. In: EDBT/ICDT'12 Workshops. pp. 239–244. ACM (2012)
11. Maule, A., Emmerich, W., Rosenblum, D.S.: Impact analysis of database schema changes. In: ICSE '08. pp. 451–460. ACM (2008)
12. Meurice, L., Bermudez, J., Weber, J., Cleve, A.: Establishing referential integrity in legacy information systems: Reality bites! In: ICSM '14. IEEE (2014)
13. Nagy, C., Meurice, L., Cleve, A.: Where was this SQL query executed?: A static concept location approach. In: SANER '15, ERA Track. IEEE (2015)
14. Ngo, M.N., Tan, H.B.K.: Applying static analysis for automated extraction of database interactions in web applications. Inf. Softw. Technol. 50(3), 160 (2008)
15. Sonoda, M., Matsuda, T., Koizumi, D., Hirasawa, S.: On automatic detection of SQL injection attacks by the feature extraction of the single character. In: SIN '11. pp. 81–86. ACM (2011)
16. Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H., Yu, J.X.: Matching dependence-related queries in the system dependence graph. In: ASE '10. pp. 457–466. ACM (2010)
17. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. ACM ToSEM 16(4) (Sep 2007)