2016 IEEE International Conference on Software Quality, Reliability and Security

Detecting and Preventing Program Inconsistencies Under Database Schema Evolution

Loup Meurice
PReCISE Research Center
University of Namur, Belgium
loup.meurice@unamur.be

Csaba Nagy
PReCISE Research Center
University of Namur, Belgium
ncsaba@inf.u-szeged.hu

Anthony Cleve
PReCISE Research Center
University of Namur, Belgium
anthony.cleve@unamur.be

***Abstract*—Nowadays, data-intensive applications tend to access their underlying database in an increasingly dynamic way. The queries that they send to the database server are usually built at runtime, through String concatenation, or Object-Relational-Mapping (ORM) frameworks. This level of dynamicity significantly complicates the task of adapting application programs to database schema changes. Failing to correctly adapt programs to an evolving database schema results in program inconsistencies, which in turn may cause program failures. In this paper, we present a tool-supported approach, that allows developers to (1) analyze how the source code and database schema co-evolved in the past and (2) simulate a database schema change and automatically determine the set of source code locations that would be impacted by this change. Developers are then provided with recommendations about what they should modify at those source code locations in order to avoid inconsistencies. The approach has been designed to deal with Java systems that use dynamic data access frameworks such as JDBC, Hibernate and JPA. We motivate and evaluate the proposed approach, based on three real-life systems of different size and nature.**

I. Introduction

Maintaining and evolving large software systems is becoming increasingly complex in the case of *data-intensive* software systems. These systems manipulate a huge amount of data usually stored in a relational database, by means of possibly complex and dynamic interactions between the application programs and the database. When the database schema evolves, developers often need to adapt the source code of the applications that accesses the changed schema elements. This adaptation process is usually achieved manually.

Furthermore, nowadays, a large variety of frameworks and libraries can be used to access the database. In particular, Object-relational mapping (ORM) technologies provide a high level of abstraction upon a relational database that allows developers to use the programming language they are comfortable with instead of using SQL statements and stored procedures. As a consequence, the interactions between the program source code and the database may become more dynamic, and thus more complex to understand. In this context, manually recovering the database access locations in the source code and precisely identifying the database elements accessed at those locations may prove complicated due to higher levels of abstraction and dynamicity. Thus, assessing the impact of a database schema change on the source code is becoming increasingly complex and error-prone for developers.

This paper addresses this particular problem. It presents a tool-supported approach to detect and prevent program inconsistencies under database schema changes. The approach analyzes the evolution history of the system in order to identify program inconsistencies due to past database schema changes. By means of a *what-if* analysis, our approach also allows developers to simulate future database schema modifications and to determine how such modifications would affect the application code. In order to ensure that the programs consistency is preserved under those schema changes, our approach makes recommendations to developers about where and how they should propagate the schema changes to the source code.

The remainder of this paper is structured as follows. Section II presents different Java database access technologies and elaborates on the difficulty to maintain a system in presence of such (possibly co-existing) technologies. In Section III, we focus on the history analysis – and we motivate the need for a what-if analysis approach – by analyzing how database schemas and programs co-evolved in three large open-source Java systems. We present our what-if analysis approach in Section IV. We then evaluate the accuracy of our approach by applying it to those three systems in Section IV-B. In Section V, we discuss the current limitations of our approach. A related work discussion is given in Section VI. In Section VII we conclude the paper and anticipate future directions.

II. Java Database Access Technologies

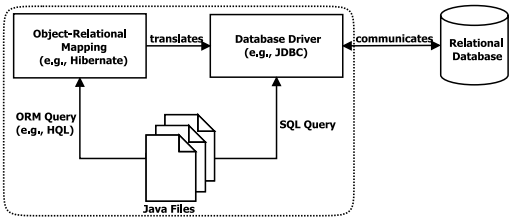

Fig. 1. Overview of the interaction between source code and relational database.

This paper focuses on the analysis of large Java systems. The choice for Java is because it is the most popular programming language today according to different sources such as the TIOBE Programming Community index [1]. In addition to this, a large-scale empirical analysis, carried out in [7],

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.38

revealed that a wide range of frameworks and APIs are used by Java projects to access relational databases. These technologies operate on different levels of abstraction. For example, a developer can simply choose to embed character strings that represent SQL statements in the source code. The SQL queries are sent to the database through a connector such as JDBC, which provides a low abstraction level of SQL-based database access. Interaction with the database can also be achieved using an ORM library. Such a library offers a higher level of abstraction based on the mapping defined between Java classes and database tables. Figure 1 illustrates such a multi-technological architecture. Our approach currently focuses on three of the most popular Java technologies (according to [7]), namely JDBC, Hibernate, and JPA. Below we briefly describe each of them together with their different database access mechanisms.

a) 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 developers a set of methods for querying the database, for instance, `Statement.execute(String)`, `PreparedStatSement.executeQuery(String)`, or `Statement.executeUpdate(String)`. Figure 2 depicts a code fragment using JDBC to execute a SQL query.

```
1  String sql = "select * from AppCustomers where id = ?";
2  PreparedStatement st = con.prepareStatement(sql);
3  st.setInt(1, cust_id);
4  ResultSet rs = st.executeQuery();
```

Fig. 2. Example of the execution of a SQL query (line 4) using JDBC.

b) Hibernate ORM: Hibernate is an object-relational mapping 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. In addition, Hibernate also provides a way to perform *CRUD operations* (Create, Read, Update, and Delete) on the instances of the mapped entity classes. Figure 3 shows an example of Hibernate code.

```
1  public List<Customer> findAllCustomers(){
2     return executeQuery("select c from Customer c");
3     }
4  public List executeQuery(String hql){
5     return session.createQuery(hql).list();
6     }
7  public Customer findCustomer(Integer id){
8     return (Customer) session.get(Customer.class, id);
9     }
```

Fig. 3. Example of the execution of a HQL query (line 5) and a CRUD operation (line 8).

c) Java Persistence API: JPA is a Java API specification to describe the management of relational data in applications. Just like Hibernate, JPA 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 provides developers with 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. Similarly to HQL, it is inspired by SQL, but operates on JPA entity objects rather than directly on database tables. JPA also provides a way to perform CRUD operations on the instances of mapped entity classes. Figure 4 shows an example of JPA code.

```
1  public Customer getCustomerByOrder(Order order){
2    Integer cust_id = order.getCustomerId();
3    String jpql = "SELECT c FROM Customer c WHERE c.custId
         =:id";
4    return (Customer) entityMgr.createQuery(jpql).
         setParameter("id", cust_id).getSingleResult();
5     }
6  public void saveOrder(Order order){
7    entityMgr.getTransaction().begin();
8    entityMgr.persist(order);
9    entityMgr.getTransaction().commit();
10   entityMgr.close();
11    }
```

Fig. 4. Example of the execution of a JPQL query (line 4) and a CRUD operation (line 8).

The use of an ORM bringing a higher level of abstraction or the co-existence of several database access technologies in the same Java system may cause some difficulties to maintain the source code. In particular, when developers apply some changes to the database schema, adapting the source code to those changes may be a complex task due to the system heterogeneity and/or the (implicit) mapping between the Java code elements and the database schema elements.

III. History Analysis for Inconsistency Detection

In this section, we present the motivation of our approach through analyzing the past of large systems and in particular, the behavior of developers when adapting programs to database schema changes. We study how source code and database schema co-evolve to estimate the related effort and difficulties arising when *manually* adapting the source code to database schema changes. This history analysis will help developers to understand how source code and database co-evolve over time. Moreover, analyzing the co-evolution history may help developers to detect program inconsistencies due to awkward past database schema changes which were not correctly propagated to the code, and to understand how the system has come thus far. By this exploratory analysis, we aim to establish the necessity and the potential benefit of a tool-supported approach helping developers to achieve future database-program co-evolution tasks.

For achieving this, we will analyze the history of three large real-life systems and in particular how developers adapted the source code to database schema changes. We decided to

target 4 types of database schema changes: deleting a table, renaming a table, deleting a column and renaming a column. We selected those types of changes since (1) they belong to the categories of changes observed in practice by several authors [3], [4], [15], [26], [29], and (2) because those types of changes could potentially break the program source code. Adding a new table, or adding a new index, for instance, while also frequently used in practice [4], do not have an immediate impact on program source code. The four types of changes we consider may potentially make existing program queries fail, in case the source code is not properly adapted.

A. Subject Systems

We considered three open-source Java systems of significant size: OpenMRS, Broadleaf Commerce and OSCAR. We selected those three systems because they have all a significant history and code size, and use different database technologies.
OpenMRS (www.openmrs.org) is a collaborative open-source project to develop software to support the delivery of healthcare in developing countries (mainly in Africa). It was conceived as a general-purpose EMR system that could support the full range of medical treatments. It has been developed since 2006. OpenMRS uses a MySQL database accessed via Hibernate and dynamic SQL (JDBC).
Broadleaf Commerce (www.broadleafcommerce.org) is an open-source, e-commerce framework written in Java on top of the Spring framework. It facilitates the development of enterprise-class, commerce-driven sites by providing a robust data model, services, and specialized tooling that take care of most of the 'heavy lifting' work. Broadleaf has been developed since 2008. It uses a relational database accessed via JPA.
OSCAR (www.oscar-emr.com) is an open-source ERM information system that is widely used in the healthcare industry in Canada. Its primary purpose is to maintain electronic patient records and interfaces of a variety of other information systems used in the healthcare industry. OSCAR has been developed since 2002. OSCAR combines different ways to access the database like JDBC, Hibernate and JPA. Table I gives some characteristics of OpenMRS, Broadleaf Commerce and OSCAR.

TABLE I
SIZE METRICS OF THE SYSTEMS - CREATION DATE, NUMBER OF VERSIONS COMMITTED IN THE VERSIONING REPOSITORY, NUMBER OF CODE LINES AND DATABASE SCHEMA SIZE.

System	Start Date	Versions	KLOC	Tables	Columns
OpenMRS	05/2006	$>$ 9100	$>$ 300	88	951
Broadleaf	12/2008	$>$ 7700	$>$ 250	179	965
OSCAR	11/2002	$>$ 21000	$>$ 2000	512	15680

B. Historical Dataset Extraction

For each of the three systems, we extracted their corresponding historical dataset. The historical dataset is the result of a process exploiting the system's history, i.e., the versioning repository. To extract the historical dataset, it is not required to exploit the whole system's history; indeed, the user can decide to concentrate on a limited set of versions. The extraction of the historical dataset consists of 3 main steps as illustrated by Figure 5. The first one (*Database Schema Extraction*) relies on an approach introduced in our previous work [15]: for each selected version, we first extract the corresponding database schema from the versioning repository. Then, by comparing each database schema version, we compute the *historical database schema*. Through this analysis step, we gain knowledge of the whole database history; we know when and how tables and columns were created/modified/deleted.

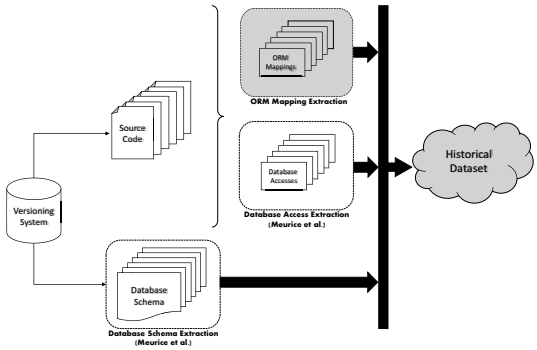

Fig. 5. Overview of our historical dataset extracted by exploiting the versioning system to mine the database schema, database accesses and ORM mappings of n system versions. This process builds on our previous work. The grey modules represent novel components.

The second step (*Database Access Extraction*) uses the tool support we developed in another previous work dedicated to database access recovery in Java source code [16]. We use this static analysis approach to automatically locate and extract all the database accesses that use JDBC, Hibernate and JPA. For each detected database access (and for each version), we extract the exact code locations where the access is executed as well as the tables/columns manipulated by the access.

The third analysis step (*ORM Mapping Extraction*) aims to detect and extract all the ORM mappings defined between a class (resp. attribute) and a table (resp. column). For achieving this, we implemented a module aiming to parse the Java code and the XML files in which the JPA/Hibernate mappings are defined. For each version, we determine where and how each table/column is mapped to the Java code elements.

The historical dataset that we obtain is organized according to the Entity-Relationship (ER) data model depicted in Figure 6. The central element of this model is the Version of the system. The historical dataset consists of the set of committed versions. A version is committed by a particular developer at a particular date and is identified by a hash value. The model is composed of 4 main different parts:

1) The source code history (green components). This part represents the history of the source code objects. A Java File may contain several Classes, Methods and Attributes. Each code object may exist in several versions and, for each version, the object has a particular position in the code (CodeObjectPosition) expressed as a couple of coordinates: a begin line and column, and an end line and column.

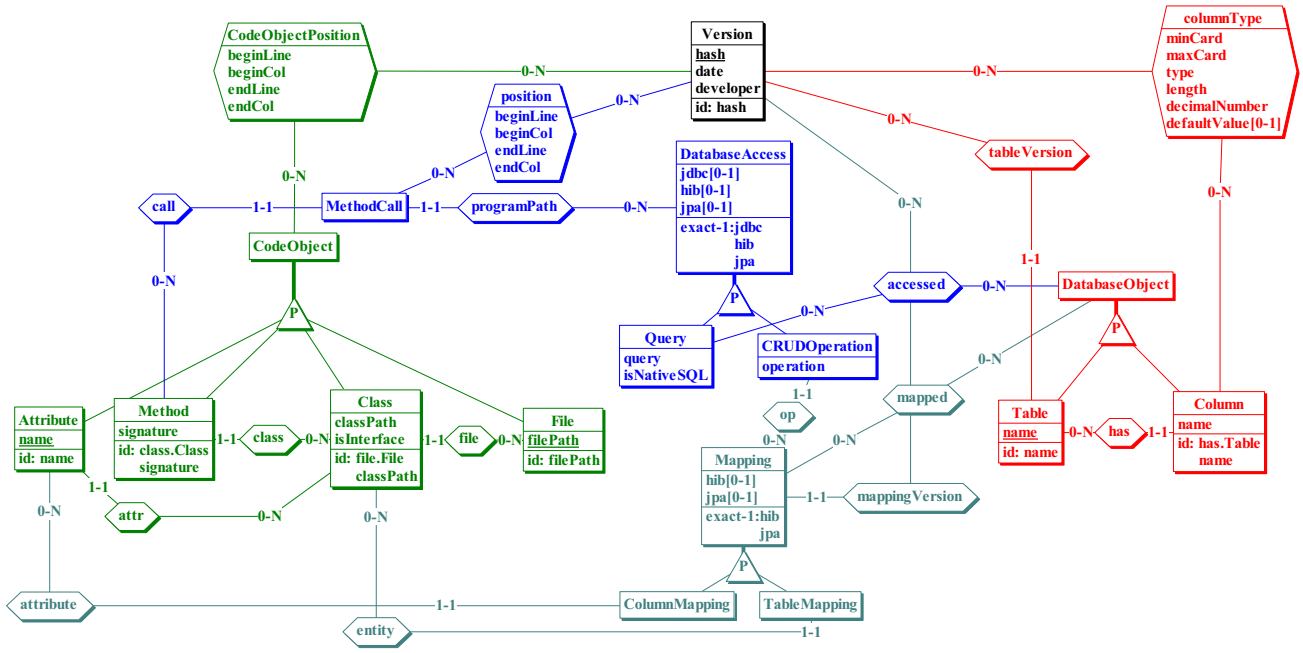

Fig. 6. ER data model of a historical dataset.

2) The database schema history (red components). The database schema evolves over time and may have a different set of schema objects (DatabaseObjects). Only schema objects of type Table and Column are considered in the model. The database tables and columns may be present in several versions. Depending on the version, a column may have a different type (columnType).

3) The ORM mapping history (grey components). By means of an ORM (e.g., Hibernate/JPA), developers can define a Mapping between an entity class and a table (TableMapping) or between an attribute and a column (ColumnMapping). An ORM mapping may exist in several versions.

4) The database access history (blue components). This part represents the history of the database accesses, i.e., of the source code locations that provide an access to the database. Those database accesses (DatabaseAccess) use a particular technology (e.g., JDBC, Hibernate or JPA) to query the database and are located at a particular position in the source code. Moreover, the creation and the execution of a database access are not always performed at the same source code location but can be achieved through successive calls to several methods. Therefore the program path (programPath) of each database access is also represented. Such a program path is the minimal set of method calls (MethodCall) needed to create and execute a database access. We represent two different kinds of accesses: Query and CRUDOperation. While the query (e.g., SQL, HQL, JPQL) is embedded in the code and allows a direct access in the form of a query string, a CRUD operation is an operation performed on the instances of the mapped entity classes. For each database access, the set of accessed database objects is recorded. A database access may, in turn, exist in several versions.

Let us illustrate those notions using Figure 3, that shows a sample of Java code accessing the database by means of Hibernate in a given version of the system. In that example, one can identify the presence of two database accesses: line 8 retrieves the customer corresponding to a given id, whereas line 2 executes an HQL query selecting all the customers recorded in the system. The first access makes use of `Customer`, a mapped entity Class located in File `/src/pojo/Customer.java`, to query the database. `Customer` is mapped (Mapping) to the `AppCustomers` Table. Line 8 is a Hibernate DatabaseAccess and more precisely a CRUDOperation (of type Read). The ProgramPath of the read access has a length of 1 and is a MethodCall to `findCustomer` Method at line 8 (position). The second database access is an HQL Query accessing the `AppCustomers` Table and has a ProgramPath of length 2: a MethodCall to the `findAllCustomers` Method at line 2 and a second call to the `executeQuery` Method at line 5.

For each of the three systems, we thus extracted the corresponding historical dataset from the respective version control repositories. First, we picked the respective initial commits and then we went on through the next versions and selected those that were at least 15 days far from the last selected version and contained several hundreds of modified lines. As a result, we have a snapshot of the state of each system every two weeks of its development. We respectively selected 164, 118 and 242 versions for OpenMRS, Broadleaf and OSCAR,

and preprocessed their respective historical dataset including all the selected versions. Each dataset follows the data model of Figure 6.

C. Analyzing the co-evolution history of three large systems

Analyzing how database schemas and programs co-evolve in Broadleaf, OpenMRS and OSCAR will help us to understand how they co-evolve over time and to establish the usefulness and the potential benefit of our what-if analysis approach which can support developers to achieve co-evolution tasks (presented in Section IV). For that, we analyze the historical datasets of the three systems in order to evaluate the effort required in the past (without our what-if analysis approach) for adapting the applications source code in reaction to database schema changes. As previously explained, we focused on 4 types of database schema changes performed in the systems history, namely deleting a table, renaming a table, deleting a column and renaming a column. We rely below on several co-evolution metrics to estimate the time and effort required to propagate database schema changes to the programs source code.

1) Deleting a table/column: For evaluating the impact of a table deletion (TD) and a column deletion (CD) on the source code, we analyzed the database schema history of each system to identify the set of tables and columns which have been deleted. We only considered the tables/columns which have been *permanently* removed from the schema; some deletions may sometimes be done by distraction and are directly recovered once identified. Tables II and III summarize the different measures we use for evaluating the co-evolution effort needed to deal with a table and column deletion at the source code level, respectively. The first column of both tables expresses the number of table/column deletions detected in each system. The second represents the number of deletions which are still currently unsolved, i.e., for which there still exist some accesses to the deleted table/column in the source code or for which an ORM mapping linking an entity class/attribute to the deleted table/column still exists. The third column shows the geometric average time (expressed in number of versions) needed to adapt the code (no more ORM mapping or access to the table/column). We use the geometric average in order to avoid to be affected by extreme values. The third column also includes the longest/maximal period of time to solve a table/column deletion in the source code (also expressed in number of versions). The minimal number of versions to solve a table/column deletion in the best case is one version: thus removing the table/column from the schema counts as one. This arbitrary choice is justified by the use of the geometric average, for which values must be different from 0. The last column indicates, for the tables/columns that were accessed before their deletion, the average and maximum numbers of related source code locations, i.e., the number of accesses or mappings that would potentially be impacted as a propagation of each table/column deletion.

The three systems present quite different figures. Deleting a table or a column in OpenMRS and OSCAR seems to be costly and tedious. By observing the database schema, one can notice developers rarely remove database objects. The general trend suggests that developers add new schema objects (much) more often than they remove existing objects. However, a table/column deletion does not come at no cost in terms of program adaptation.

TABLE II
CO-EVOLUTION METRICS RELATED TO TABLE DELETIONS.

System	#Table Deletions	#Unsolved	Propagation Time avg∼max	#Accesses avg∼max
OpenMRS	11	1	1.6 ∼ 134	7.5 ∼ 9
Broadleaf	86	0	1.1 ∼ 6	2.8 ∼ 14
OSCAR	33	5	1.4 ∼ 90	2.9 ∼ 9

TABLE III
CO-EVOLUTION METRICS RELATED TO COLUMN DELETIONS.

System	#Column Deletions	#Unsolved	Propagation Time avg∼max	#Accesses avg∼max
OpenMRS	32	4	1.6 ∼ 134	2.2 ∼ 4
Broadleaf	154	0	1.1 ∼ 2	4 ∼ 15
OSCAR	170	0	1.1 ∼ 24	1.6 ∼ 132

For OpenMRS, if the deleted table was accessed, up to 9 source code locations could be impacted, 7.5 locations on average. For a deleted column, up to 4 code locations could be impacted, 2.2 locations on average. Moreover, some deletions remain unsolved and there still exist some code locations accessing the deleted table/column. For instance, the deletion of the `FORM_RESOURCE` table has never been propagated to the source code up to now. The deletion happened in October 2011 but the developers omitted to delete an old Hibernate mapping still currently existing[1]. Through this older mapping, OpenMRS still offers an interface to access the removed table. We observed the same trend for some column deletions too[2].

Moreover, by analyzing the geometric average and maximal time necessary to solve a table/column deletion, we observe that removing a table/column is far from being trivial; on average, almost 2 versions are needed (1 version = 15 days), while the most costly deletion took 134 versions[3]. Another interesting point in OpenMRS is that it seems that all developers are not always aware of a table/column deletion. We found several deletions which had not been considered by some developers who have continued to create new accesses to the removed tables/columns. Those accesses have been dropped only after 76 versions on average and some of them are still present in the source code at the time of writing this paper.

OSCAR developers also seem to face some difficulties with table/column deletions. Some table deletions have never been propagated to the source code up to now. There still exist some ORM mappings and code locations allowing developers to access the deleted table. While the average time to propagate

[1]You can find in [29] the proof of the existence of that mapping in April 2015. The database schema of that version can be found in [30].

[2]The `VOIDED` column has been removed from the `USERS` table in September 2011 and is still currently accessed [33] via a JDBC query.

[3]The `REPORT` table and all its columns were deleted in May 2008. However, OpenMRS still accessed it until August 2010 [31] and defined a Hibernate mapping until April 2014 [32].

the table/column deletions seems quite short (1,4 and 1,1 version), the maximal values (90 and 24 versions) illustrate again that the propagation process is far from being trivial. Furthermore, we also found 6 table deletions which had not been considered by some developers who have continued to create new accesses to the removed tables. Those accesses have been dropped only after 34 versions on average.

Broadleaf developers seem to better propagate table/column deletions than OpenMRS and OSCAR developers. They have had more deletions to achieve, but with less impacted source locations on average. On average, it only took them 1.1 version to adapt the source code to a deletion. However, by observing the maximal values, one notices that the propagation process is not always straightforward. In contrast to OpenMRS and OSCAR, all developers seem to be aware of each deletion since we did not observe the creation of *post-mortem* accesses.

2) Renaming a table/column: For evaluating the impact of a table renaming (TR) and a column renaming (CR), we only focused on those tables/columns which have been renamed on purpose. Tables IV and V shows the co-evolution metrics we used for each system (respectively for the table and column renamings). The first column shows the number of renamed tables/columns in each system. Like for the deletions, we calculated the average and maximal number of versions necessary for the renamed table/column to be *solved*, i.e., there is no more access and ORM mapping to the renamed table/column (columns 2 and 3). During a table/column renaming phase, the developers try to co-evolve the code in order to adapt the outdated database accesses to the new table/column name.

TABLE IV
CO-EVOLUTION METRICS RELATED TO TABLE RENAMINGS.

System	Renaming	Solution Time avg∼max	#Accesses avg∼max
OpenMRS	1	1 ∼ 1	0 ∼ 0
Broadleaf	14	2.6 ∼ 6	3.3 ∼ 8
OSCAR	7	1.9 ∼ 89	1.6 ∼ 132

TABLE V
CO-EVOLUTION METRICS RELATED TO COLUMN RENAMINGS.

System	Renaming	Solution Time avg∼max	#Accesses avg∼max
OpenMRS	10	1 ∼ 1	0 ∼ 0
Broadleaf	16	1.1 ∼ 2	1.7 ∼ 3
OSCAR	321	1.2 ∼ 38	1.7 ∼ 389

As a matter of fact, OpenMRS does not constitute a suitable system to study the impact of a table/column renaming on the code. Indeed, only one table and ten columns have been renamed in the past and have been immediately solved. The renamed table/columns were not part of a Hibernate mapping and were never accessed in the source code. The programs started to access it a few versions after its renaming.

In OSCAR, 7 tables have been renamed, with an average of 1.6 impacted locations per table renaming (despite one extreme value of 132 locations). However, OSCAR developers have considerably more renamed the columns, with 321 column renamings. On average, 1.7 locations are impacted by a column renaming, despite an extreme value of 389 locations. However, one can notice that renaming is a tedious and costly refactoring that may have unintentional impact on the code for a longer period; some table and column renamings took several years to be fully propagated to the code[4].

In Broadleaf, 14 tables have been renamed, which significantly impacted the code (up to 8 impacted locations per table renaming). This impact is mainly due to the developers' strategy to rename the Java entity class mapped to the renamed table in order to better fit with the new name. Therefore, changing the current JPA annotation is not enough to deal with a table renaming and modifying each access is thus necessary. On average, 2.6 versions are required to remove the JPA mapping/accesses to the renamed tables. Furthermore the most costly renaming took 6 versions to be propagated to the source code.

Broadleaf's developers also rename *active* columns, as shown by the number of accesses to the renamed columns. Most columns were accessed before their renaming, and the propagation of the renamings to the code was immediate. However, such a quick reaction can be easily explained. All related accesses rely on JPA, therefore editing the JPA annotation is sufficient to propagate the column renaming[5].

In summary, we proposed a first approach allowing us to analyze the co-evolution history of OpenMRS, Broadleaf and OSCAR. We made several interesting observations; we noticed that source code adaptation is not always a trivial task. We observed that a schema change may require several months before the source code is adapted and sometimes, those schema changes cause outdated database accesses which are never adapted and which could break the code. Even worse, in some cases, developers keep creating new accesses to removed or renamed schema objects. It allowed us to point out query fails related to awkward past changes.

IV. WHAT-IF ANALYSIS FOR CONSISTENCY PRESERVATION

Through the history analysis of those systems, we motivated the need for an automated what-if analysis approach helping developers to simulate hypothetical database schema evolutions and determine their impact on the source code. The objectives of our what-if analysis approach are (1) to facilitate database-program co-evolution by determining the source code locations impacted by a database schema change and (2) to ensure that the system consistency is preserved over time under (successive) schema changes.

[4]The `FORMCOUNSELLORASSESSMENT` table was renamed in December 2006 but was still accessed in September 2010 [36].

[5]We illustrate an example of a JPA annotation modification to deal with the renaming of a column. The `DATE` column has been renamed as `DATE_RECORDED` in April 2013. You can find the JPA annotation before [34] and after [35] the renaming.

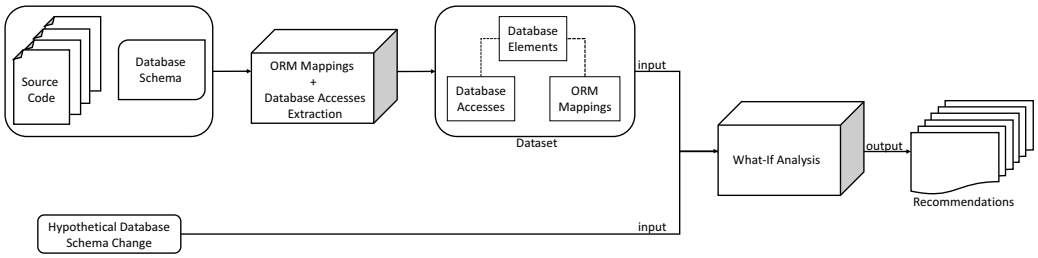

Fig. 7. Overview of our what-if analysis approach.

A. Approach

Our approach allows developers to simulate a database schema change and provides them with related recommendations on how to adapt the application's source code to that change. It tackles the issue of adapting the source code when the database schema has been modified. The objective of our approach is to propose an answer to the question "**where** and **how** should I change the code if I apply this particular database schema change?".

Figure 7 summarizes our approach. It takes 2 inputs, namely (1) a given version of the system code (e.g., the current system's version), and (2) a hypothetical database schema change (e.g., *I wish to delete table t.)*. We compute the former in order to obtain a dataset with a structure close to the ER model (Figure 6), including information related to the database accesses and ORM mappings but without the notion of *versions* (since the process is applied to a single version). The result of our what-if analysis approach is a list of recommendations made to developers for adapting the code to that change (e.g., *You need to remove the Hibernate mapping defined between table t and Java class c at this location.)*. Each recommendation invites the developer to modify a particular source code location which would be impacted by the future database schema change.

Let o be the database object to modify in the schema, A_o, the set of code locations accessing o and M_o, the set of ORM mappings referencing o. A_o is defined as $A_o = A_o^e \cup A_o^i$; where A_o^e and A_o^i represent the sets of code locations that, respectively, *explicitly and implicitly*[6] access o.

Once all those sets are computed, we know the code locations potentially impacted by a future modification of o. Depending on the type of the operation to perform on o, the impact on the code is different. This is why we propose a strategy to deal with 5 types of database schema changes: deleting/renaming a table, deleting/renaming a column and changing the type of a column. We selected those types of changes since (1) they belong to the categories of changes observed in practice by several authors [3], [4], [15], [26], [29], and (2) because those types of changes potentially have an impact on program source code. Adding a new table, or adding a new index, for instance, while also frequently used in practice [4], do not have an immediate impact on program source code. The 5 types of changes we consider may potentially make existing program queries fail, in case the source code is not properly adapted.

Each strategy is summarized in Table VII. A strategy is composed of *recommendations* or *warnings* provided to the user. A recommendation indicates a *mandatory* modification to apply to a particular source code location; otherwise it would be *broken* by the database schema change. A warning invites the developer to pay attention to a particular source code location which *might* require a modification; the developer should thus manually inspect the detected code location to verify if any modifications are actually required.

Let us illustrate the use of our strategy table through a concrete example. The developers of a system foresee a future database schema change but first they want to estimate the cost of that change by assessing the impact on the code with our what-if analysis approach. They wish to rename the $CUST$ table (as $CUSTOMER$) as well as alter the type of the $POSTAL\_CODE$ column (integer to string). Figure 8 depicts a piece of the source code before (left) and after (right) applying the recommendations made by our approach. The latter automatically computes A_{CUST}, M_{CUST}, A_{POSTAL_CODE} and M_{POSTAL_CODE} (see Table VI). The developers cope with two database schema changes:

1) Renaming the $CUST$ table: (1) the JPA annotation (line 2) is renamed (ID = **III**) and (2) the SQL query (line 20) is adapted to the new table name (ID = **IV**)

2) Changing the type of the $POSTAL\_CODE$ column: (1) the JPA attribute (line 9) type is modified (ID = **VI**), (2) the equality condition `c.postalCode = code` of the SQL query (line 27) is modified by the adding of apostrophes to fit with the new string type (ID = **VII**) and finally, by inspecting the code locations (recommended by our approach), the developers could have spotted the affected locations (lines 15, 16, 22 and 26) and corrected them (ID = **V**).

B. Evaluation

In this section, we assess the accuracy of our what-if analysis approach. This evaluation aims to measure (1) *correct*

[6]The `ID` column is explicitly accessed by the following SQL query, while all the other columns of `CUSTOMER` are implicitly accessed: `select * from CUSTOMER where ID = 0`.

TABLE VI

THE CODE LOCATIONS ACCESSING THE CUST TABLE AND THE POSTAL_CODE COLUMN DETECTED BY OUR WHAT-IF ANALYSIS APPROACH.

	o = CUST	o = POSTAL_CODE
M_o	• [L2]`@Table(name= "CUST")`	• [L9]`@Column(name= "POSTAL_CODE")`
A_o^e	• [L20]`SELECT * FROM cust WHERE customer_id = id` • [L27]`from Customer c where c.postalCode = code`	• [L27]`from Customer c where c.postalCode = code`
A_o^i		• [L20]`SELECT * FROM cust WHERE customer_id = id`

TABLE VII

STRATEGIES (RECOMMENDATIONS AND WARNINGS) FOR FACING A DATABASE SCHEMA CHANGE.

Operation	Strategy	Type	ID
Deleting table	• Deleting all the ORM mappings of M_o	Recommendation	I
	• Modifying/Deleting all the accesses of A_o^e	Recommendation	II
Renaming table	• Modifying the table in each mapping of M_o. For instance, a JPA mapping will be modified by changing the table name in the annotation.	Recommendation	III
	• Modifying the SQL queries of A_o^e. Indeed, while the Hibernate/JPA accesses are not impacted, the SQL queries have to be adapted to the new table name.	Recommendation	IV
Deleting column	• Deleting all the ORM mappings of M_o	Recommendation	I
	• Modifying/Deleting all the explicit accesses of A_o^e.	Recommendation	II
	• Inspecting the code locations of A_o^e and A_o^i. The value of an accessed column may be explicitly used in the code. In such a case, our what-if analysis approach proposes the developer to further inspect the code locations accessing o to ensure that the value of the deleted column is not used later in the code.	Warning	V
Renaming column	• Modifying the column in each mapping of M_o	Recommendation	III
	• Modifying all the SQL queries of A_o^e	Recommendation	IV
	• Inspecting the code locations executing an access of A_o^e and A_o^i. The approach proposes the developer to further inspect the code locations accessing o to ensure that the value of the renamed column is not used later in the code.	Warning	V
Changing column type	• Changing the type of all mapped attributes in M_o	Warning	VI
	• Inspecting the accesses of A_o^e to ensure that the column value is not used in an equality condition or in an assignment statement. If needed, modify this condition/assignment to comply with the new type.	Warning	VII
	• Inspecting the code locations executing an access of A_o^e and A_o^i. The approach proposed the developer to further inspect the code locations accessing o to ensure that the value of the column, if used in the code, is stored in a well-typed variable.	Warning	V

```java
@Entity
@Table(name = "CUST")
public class Customer{
  @Id
  @GeneratedValue(generator = "AddressId")
  @Column(name = "CUSTOMER_ID")
  protected Long id;

  @Column(name = "POSTAL_CODE", nullable = false)
  protected int postalCode;
                ...
}

public class CustomerDAO {
  public int getPostalCodeByCustId(Long id){
    int postalCode;
                        ...
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(
    "SELECT * FROM cust WHERE customer_id="+id);
    if(rs.next())
      postalCode = rs.getInt("postal_code");
                        ...
    return postalCode;
  }
  public List<Customer> getPostalCusts(int code){
        String hql = "from Customer c where c.postalCode = " +
              code;
        List<Customer> list = session.createQuery(hql).list();
        return list;
  }
}
```

```java
@Entity
@Table(name = "CUSTOMER") // ID=III
public class Customer{
  @Id
  @GeneratedValue(generator = "AddressId")
  @Column(name = "CUSTOMER_ID")
  protected Long id;

  @Column(name = "POSTAL_CODE", nullable = false)
  protected String postalCode; // ID=VI
                ...
}

public class CustomerDAO {
  public String /*ID=V*/ getPostalCodeByCustId(Long id){
    String postalCode; // ID=V
                        ...
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(
    "SELECT * FROM customer WHERE customer_id="+id); // ID=IV
    if(rs.next())
      postalCode = rs.getString("postal_code"); // ID=V
                        ...
    return postalCode;
  }
  public List<Customer> getPostalCusts(/*ID=V*/String code){
    String hql = "from Customer c where c.postalCode = '" +
          code + "'"; // ID=VII
    List<Customer> list = session.createQuery(hql).list();
    return list;
  }
}
```

Fig. 8. Java code before (left) and after (right) the co-evolution with the help of our what-if analysis approach.

recommendations, (2) *wrong recommendations* and (3) *missing recommendations*. For calculating those metrics, we rely on the history of *Broadleaf*, *OpenMRS* and *OSCAR*. Among the whole set of database schema changes that we observed in the life of those systems (855 changes), we first selected a subset of changes that would be sufficiently relevant to assess: we only considered the database schema changes performed on a database object (table/column) which was still active and used in the applications' code before the schema modification (i.e., database object concerned by an ORM mapping or accessed somewhere in the code). Moreover, we decided not to include the column type changes in the evaluation; the strategy defined in Table VII to deal with the column type changes is exclusively composed of warnings and therefore, we excluded them. By applying those selection conditions, we obtained a subset of 323 database schema changes. We then randomly selected 130 changes, which represent about 40% (130/323) of all schema changes with potential impact on the code. Table VIII shows the distribution of those changes.

TABLE VIII
DISTRIBUTION OF THE 130 SELECTED DATABASE SCHEMA CHANGES.

	TR	TD	CR	CD
Broadleaf	12	17	12	52
OpenMRS	0	2	0	5
OSCAR	5	9	7	9
Total	**17**	**28**	**19**	**66**

For each of those 130 changes, we applied our what-if analysis approach to the system version before the schema change and obtained a set of recommendations/warnings. We then manually calculated the number of:

1) *Correct recommendations*: the recommendations which were (and/or should have been) actually followed by the developers after the schema change.

2) *Wrong recommendations*: the recommendations which were not (and/or should not have been) actually followed by the developers after the schema change.

3) *Missing recommendations*: the modifications actually applied to the code which constitute a correct propagation of the schema change, but were not recommended by our approach. For detecting those missing recommendations, we manually analyzed the code locations directly linked to the modified database objects (e.g., the accesses, ORM mappings).

While a warning proposed by our approach represents an advice for the developers to manually inspect if any changes are required, it might constitute a *soft* warning and could be ignored by the developers. Therefore, we do not consider an ignored warning as a wrong recommendation. However, we consider an actually followed warning as a correct one.

Table IX presents the results of our manual evaluation. Out of 130 schema changes, our what-if analysis approach proposed 204 recommendations: 99% are correct recommendations, while only 1% constitute wrong recommendations. Those wrong recommendations come from the deletions of two columns. Actually, these columns were not removed but moved to another table. Since our approach considered those changes as deletions, it generated two wrong recommendations pertaining to the deletion of the linked ORM mappings. In the future, we expect to extend our what-if analysis in order to deal with column move operations. Among the 130 schema changes, only 5% of them (6/130) missed a recommendation. Indeed, some ORM mappings are not detected by our code analyzer and are missed in the resulting recommendations.

TABLE IX
RATES OF CORRECT, WRONG AND MISSING RECOMMENDATIONS.

	TR	TD	CR	CD	Total	Perc.
Correct recommendations	17	90	24	71	**202**	**99%**
Wrong recommendations	0	0	0	2	**2**	**1%**
Missing recommendations	2	0	1	3	**6**	**5%**

Replication. All our evaluation results are available via our companion website at https://staff.info.unamur.be/lme/QRS2016/evaluation/. In particular, the reader can inspect each of the 130 assessed schema changes and verify the validity of our evaluation. For each schema change, one summarizes the recommendations made by our what-if analysis tool. Each recommendation is systematically checked against the actual source code modifications. Direct links to the related source code locations before and after the propagation of the schema changes are provided, so that the validity of our manual classification (correct/wrong/missing) can be cross-checked. An interactive demo of our what-if analysis approach is also available via our companion website available at https://staff.info.unamur.be/lme/QRS2016/play. For recent versions of OpenMRS, Broadleaf and OSCAR, the user can select a database schema object and simulate a schema change operation. The website returns related recommendations, including links to the impacted source code locations on GitHub.

V. LIMITATIONS

In this section, we discuss the current limitations of our co-evolution history analysis and our what-if analysis approach, some of them potentially affecting our evaluation results.

A. Database Access Extraction

In our co-evolution history analysis (Section III) and what-if analysis (Section IV), we use the tool support we developed in our previous work [16] to extract the database accesses from the Java source code. With this tool support, we are able to identify which portion of the source code accesses which portion of the database. Unless that tool dedicated to Java systems is compatible with JDBC, Hibernate and JPA, it also suffers from some limitations. Indeed, the access extraction step is a static analysis which constructs an abstract syntax tree and uses a visitor to navigate through the different Java nodes and expressions. By navigating through the tree, the tool recovers all the possible string values of each detected JBDC/Hibernate/JPA queries. However our static analyzer has also some limitations, as discussed in [16], which may affect our results:

1) Non-existent queries: our database access extractor is designed to rebuild all the possible string values for the SQL query. Thus, it considers all the possible program paths. Since it is currently unable to resolve a boolean condition (a dynamic analysis would be preferable), these cases generate some noise (false positive queries). Figure 9 illustrates an example of false positive queries from OpenMRS. Since our what-if analysis approach is based on the recovered SQL queries, it may in turn generate some wrong recommendations. Note that we did not encounter this problem by evaluating the past 130 database schema changes.

```
1 String hql = "";
2 if (isNameField)
3     hql += "select concept";
4 hql += " from Concept as concept";
5 if (isNameField)
6     hql += " where concept.shortName = '0'";
7 Query query = session.createQuery(hql);
8 return (List<Concept>) query.list();
```

Fig. 9. Example of the execution of a HQL query (line 8) and the extraction of false positive queries due to 2 identical boolean conditions. 4 possible HQL queries are extracted by our analyzer: (1) `from Concept as concept`, (2) `select concept from Concept as concept`, (3) `from Concept as concept where concept.shortName = '0'` and (4) `select concept from Concept as concept where concept.shortName = '0'`; (2) and (3) are both false positive queries.

2) Missing queries: some queries cannot be fully recovered by our analyzer due to its static nature. In [16], we discuss such a limitation and give as illustration the use of StringBuilder/StringBuffer Java objects to create a string query which are not dealt by our analyzer. Similarly, executed SQL queries sometimes include input values given by the application users. This is the case in highly dynamic applications. Thus, the static recovery of the associated SQL queries may be incomplete or missing.

However, despite its limitations, our static analyzer reached good results in the evaluation conducted in [16]: we could extract queries for 71.5%-99% of database accesses with 87.9%-100% of valid queries.

B. Deletion or Renaming?

Another limitation which might slightly affect the results of our co-evolution history analysis is the detection of table/column renamings and deletions. When we compute the *historical database schema* (first step of the historical dataset extraction described in Section III-B), we compare successive database schema versions. However, during the historical databaset schema extraction of OSCAR, Broadleaf and OpenMRS, we could not make use of SQL migration scripts between successive database versions. The unavailability of such migration scripts makes the detection of table/column renamings more complicated. Indeed, if table A is renamed as table B, there is no direct way to detect it and then by default our analyzer would consider that table A has been dropped while table B has been created without keeping a link between both tables. In order to mitigate this risk, we reused our automated support for implicit renaming detection presented in [15]. This support uses linear programming (LP), a specific case of mathematical programming allowing one to achieve the best outcome in a given mathematical model. We obtained a list of potential table and column renamings that we then manually validated/rejected. However, this technique may have missed some renamings and in our co-evolution history analysis, we might still consider that table A was dropped and table B was independently added. Nevertheless, we believe that this silence could have only slightly affected the conclusions of our historical analysis.

C. Database Schema Changes

In our what-if analysis approach, we decided to target 5 types of database schema changes: deleting a table, renaming a table, deleting a column, renaming a column and changing the type of a column. As explained, we focused on those types of changes since they seem to be the database schema changes the most likely to make existing queries fail, in case the source code is not properly adapted. However, other database schema changes could be considered in the future. For instance, creating or updating a foreign key can cause program inconsistencies if the referential integrity constraint is not satisfied by an executed query. In the future, we plan to extend the scope of our approach to other types of database schema changes (adding/updating foreign keys, merging/splitting tables, moving columns, etc.).

D. Dead Code

During the history analysis of Broadleaf, OpenMRS and OSCAR, we observed some schema changes causing outdated database accesses and which are never fixed; some code locations still allow developers to access removed or renamed schema objects. However, in our analysis, we did not systematically verify if such code locations actually represented dead code or were still reachable during the execution. However, we argue that even dead code accessing outdated schema objects has to be cleaned/fixed to make the program consistent. In the future, we plan to distinguish *active* and *dead* code locations when generating recommendations.

VI. Related Work

While the database schema evolution literature is very large [22], researchers have only recently started to pay more attention to the analysis of the *co-evolution* of database schema and application code [6], [7], [11], [12], [21].

It has been already shown that the *evolution history of the source code* can provide valuable data for *history-based recommenders*. For instance, Ying *et al.* [30] and Zimmermann *et al.* [31] have independently developed different approaches that use association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code. Potential uses of the *schema evolution history* were shown by Sjøberg [26] already in 1993. He studied the schema evolution history of a large-scale medical application and showed that even a small change to the schema may have

major consequences for the rest of the application code. Curino *et al.* [4] presented a study on the structural evolution of the Wikipedia database, with the aim to extract both a micro-classification and a macro-classification of schema changes. Vassiliadis *et al.* [29] studied the evolution of individual database tables over time in 8 different software systems. They also tried to determine whether the well-known Lehman's laws of software evolution also hold for database schema evolution. Their results [27] show that the essence of the Lehmans laws holds in this context, but that specific mechanics significantly differ when it comes to schema evolution.

Some researchers have also *studied* the evolution of database schema *in combination* with source code evolution. Goeminne *et al.* [6] empirically analyzed the evolution of the *usage* of SQL, Hibernate and JPA in a large and complex open source information system implemented in Java. Interestingly, they found that the practice of using embedded SQL is still common today. In a more recent work [7], they carried out a coarse-grained historical analysis of the usage of Java relational database technologies on 3,707 open source projects. They investigated the frequent co-occurrences of database access technologies and observed that some combinations of technologies appeared to complement and reinforce one another. Their results further motivate the need for a what-if analysis approach able to deal with multiple access technologies. Qiu *et al.* [21] empirically analyzed the co-evolution of relational database schemas and code in ten open-source database applications from various domains. They studied specific change types inside the database schema and *estimated* the impact of such changes on PHP code. Karahasanoić [11] studied how the maintenance of application consistency can be supported by identifying and visualizing the impact of changes in evolving object-oriented systems, including changes originating from a database schema. However, he focused on object-oriented databases rather than relational databases. Lin *et al.* [12] investigated how application programs and database management systems in popular open source systems (Mozilla, Monotone) cope with database schema changes and database format changes. They introduced the term 'collateral' evolution and observed that it can lead to potential problems when the evolution of an application is separated from the evolution of its persistent data, or from the database.

Using *what-if analysis* [8] for changes that occur in the schema/structure of the database was proposed by Papastefanatos *et al.* [18]–[20]. They presented Hecataeus, a framework that allows the user to anticipate hypothetical database schema evolution events and to examine their impact over a set of *queries and views* provided as input *by the user*. Unlike our approach, Hecataeus does not assess the impact at the *source code* level and does not consider the presence of *different* database access technologies.

Our approach is also closely related to *impact analysis* approaches for database schema changes, as we identify parts of the source code impacted by database schema modifications. In this domain, Maule *et al.* [14] proposed an impact analysis approach for schema changes. They studied a commercial object-oriented content management system and statically analyzed the impact set of relational database schema changes on the source code. They implemented their approach for the ADO.NET (C#) technology. Liu *et al.* [13] proposed an approach to extract the attribute dependency graph out of a database application from its source code by using static analysis. Their purpose was to aid maintenance processes, particularly impact analysis. They implemented their approach for PHP-based applications. Gardikiotis and Malevris [5] introduced a two-folded impact analysis based on slicing techniques to identify the source code statements affected by schema changes and the affected test suites concerning the testing of these applications. In contrast to these approaches, our goal is not only to calculate a complete impact set, but to use this information to provide developers with program adaptation recommendations. Our approach was designed for Java and it can handle database accesses through JDBC, Hibernate and JPA. It can also analyze heterogeneous systems, where *several* of those database access technologies *co-exist* within the same version.

Several previous papers identify, extract and analyze database *usage* in application programs. The purpose of these approaches ranges from error checking [9], [28], SQL fault localization [2], to fault diagnosis [10].

Finally, recent approaches and studies have focused on the evolution of NoSQL databases. In [24], the authors describe a framework controlling schema evolution in NoSQL applications. Scherzinger *et al.* [25] present a model checking approach to reveal scalability bottlenecks in NoSQL schemas. Ringlstetter *et al.* [23] analyzed how developers evolve NoSQL document stores by means of evolution annotations. They discovered that those annotations are actually used for other tasks than schema evolution.

As shown in Figure 7, this paper builds on our previous work. We introduced DAHLIA [15], a tool to analyze and visualize the evolution of a database schema. We used this tool for conducting a case study on the OSCAR system [3]. Later on, we presented an approach [16] allowing developers to automatically identify the source code locations accessing given database tables and columns; we used that approach in an ERA track paper [17], to locate the source code origin of a (buggy) SQL query executed at the database side. This technique analyzes a *single version* of the system, without considering its history.

VII. Conclusions

We presented a tool-supported approach supporting developers in co-evolving databases and programs in a consistent manner. The approach, particularly designed for data-intensive Java systems, aims to *identify* inconsistencies due to database schema changes by analyzing the system evolution history, and to *prevent* inconsistencies to arise by providing developers with change propagation recommendations.

We motivated the need for our approach by analyzing the co-evolution history of the three open source systems. We observed that the task of manually propagating database

schema changes to the programs source code is not always trivial. We saw, among others, that some database schema changes may require several versions to be fully propagated to the source code. We could even find schema changes that have never been (fully) propagated.

To overcome such problems to occur, we propose a what-if analysis approach, that takes as input a given version of the system and an hypothetical database schema change. Based on these inputs, it gives developers recommendations on how to propagate the input schema change to the programs. The recommendations include the exact source code locations that would be impacted by the schema change. The approach is able to deal with multiple (co-existing) database access technologies, namely JDBC, Hibernate and JPA.

We evaluated the accuracy of the returned recommendations by systematically checking them against the actual co-evolution history of three Java open-source systems. The results of this manual evaluation are very promising. The what-if approach reached 99% of correct recommendations when applied to a randomly selected, yet significant subset of schema changes.

As future work, we intend to extend the scope of our what-if analysis approach, by considering a larger set of database schema changes. In particular, we target schema changes that would require the adaptation of the program *behaviour*, such as adding uniqueness or referential constraints. Second, we plan to contribute to (partially) automate the schema change propagation process itself, via source code transformation techniques. Last but not least, we aim to generalize our work to other programming languages and database platforms.

REFERENCES

[1] Tiobe programming community index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2015-11-01.

[2] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold. Localizing SQL faults in database applications. In *Proc. of ASE '11*, pages 213–222. IEEE Comp. Soc., 2011.

[3] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97:113–121, 2015.

[4] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in Wikipedia - toward a web information system benchmark. In *Proc. of ICEIS*, pages 323–332, 2008.

[5] S.. Gardikiotis and N. Malevris. A two-folded impact analysis of schema changes on database applications. *International Journal of Automation and Computing*, 6(2):109–123, 2009.

[6] M. Goeminne, A. Decan, and T. Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In *CSMR-WCRE '14*, pages 353–357, 2014.

[7] M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in Java projects. In *Int'l Conf. Software Maintenance and Evolution (ICSME)*, 2015.

[8] M. Golfarelli, S. Rizzi, and A. Proli. Designing what-if analysis: Towards a methodology. In *Proc. of DOLAP '06*, DOLAP '06, pages 51–58. ACM, 2006.

[9] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. of ICSE '04*, pages 645–654. IEEE Comp. Soc., 2004.

[10] M. A. Javid and S. M. Embury. Diagnosing faults in embedded queries in database applications. In *Proc. of EDBT/ICDT'12 Workshops*, pages 239–244. ACM, 2012.

[11] A. Karahasanović. *Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation*. PhD thesis, University of Oslo, 2002.

[12] D.-Y. Lin and I. Neamtiu. Collateral evolution of applications and databases. In *Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, pages 31–40. ACM, 2009.

[13] K. Liu, H. B. K. Tan, and X. Chen. Aiding maintenance of database applications through extracting attribute dependency graph. *J. Database Manage.*, 24(1):20–35, January 2013.

[14] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *Proc. of ICSE 2008*, pages 451–460, 2008.

[15] L. Meurice and A. Cleve. DAHLIA: A visual analyzer of database schema evolution. In *CSMR-WCRE '14*, pages 464–468, 2014.

[16] L. Meurice, C. Nagy, and A. Cleve. Static analysis of dynamic database usage in java systems. In *Proc. of CAiSE '16*, LNCS. Springer, 2016. to appear. https://staff.info.unamur.be/lme/CAISE16/MeuriceEtAl.pdf.

[17] C. Nagy, L. Meurice, and A. Cleve. Where was this SQL query executed?: A static concept location approach. In *Proc. of SANER 2015, ERA Track*. IEEE Comp. Soc., 2015.

[18] G. Papastefanatos, F. Anagnostou, Y. Vassiliou, and P. Vassiliadis. Hecataeus: A what-if analysis tool for database schema evolution. In *Proc of CSMR '08*, pages 326–328, April 2008.

[19] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. What-if analysis for data warehouse evolution. In IlYeal Song, Johann Eder, and ThoManh Nguyen, editors, *Data Warehousing and Knowledge Discovery*, volume 4654 of *LNCS*, pages 23–33. Springer, 2007.

[20] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. Hecataeus: Regulating schema evolution. In *Proc of ICDE 2010*, pages 1181–1184, March 2010.

[21] D. Qiu, B. Li, and Z. Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Joint European Software Engineering Conf. and ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. ACM , 2013.

[22] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.

[23] A. Ringlstetter, S. Scherzinger, and T. F. Bissyandé. Data model evolution using object-nosql mappers: Folklore or state-of-the-art? In *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*, BIGDSE '16, pages 33–36, New York, NY, USA, 2016. ACM.

[24] S. Scherzinger, T. Cerqueus, and E. C. d. Almeida. Controvol: A framework for controlled schema evolution in nosql application development. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1464–1467, April 2015.

[25] S. Scherzinger, E. De Almeida Cunha, F. Ickert, and M. Del Fabro Didonet. On the necessity of model checking nosql database schemas when building saas applications. In *Proceedings of the 2013 International Workshop on Testing the Cloud*, TTC 2013, pages 1–6, New York, NY, USA, 2013. ACM.

[26] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35 – 44, 1993.

[27] I. Skoulis, P. Vassiliadis, and A. Zarras. Open-source databases: Within, outside, or beyond lehmans laws of software evolution? In *CAISE '14*, volume 8484 of *LNCS*, pages 379–393. Springer, 2014.

[28] 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 *Proc. of SIN '11*, pages 81–86. ACM, 2011.

[29] P. Vassiliadis, A. V. Zarras, and I. Skoulis. How is life for a table in an evolving relational schema? birth, death and everything in between. In *Proc. of ER 2015*, pages 453–466, 2015.

[30] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, September 2004.

[31] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[32] http://bit.ly/1XzyuWu.

[33] http://bit.ly/24aa3DQ.

[34] http://bit.ly/1OhJYqu.

[35] http://bit.ly/1Q39but.

[36] http://bit.ly/1Q39ipQ.

[37] http://bit.ly/1VpsZIF.

[38] http://bit.ly/1oMgEE3.

[39] http://bit.ly/1RkAn7w.