Analyzing, Understanding and Supporting the Evolution of Dynamic and Heterogeneous Data-Intensive Software Systems

Loup Meurice

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Publicly defended on 22^{nd} June 2017 in Namur (Belgium)

University of Namur

PReCISE Research Center

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Graphisme de couverture: © Loup Meurice

Imprimé en Belgique
ISBN : 978-2-87037-988-2
Dépôt légal: D/2017/1881/29

ABSTRACT

Nowadays, information systems represent crucial assets in most enterprises, since they support the majority of their business activities. Those systems are usually large software systems that manipulate a large amount of data, hence the name *data-intensive software system* (DISS). A DISS is generally composed of a collection of application programs which intensively interact with a database. The goal of the database is to collect all the relevant data about the application domain of the system.

A DISS is subject to continuous modification due to changes in the environment in which it operates. DISS evolution is an indispensable process to keep systems adapted to ever-changing business needs and technological platforms. During this process, any change in the business requirements necessitates the synchronized adaptation of the database and the programs. However, database and program source code are generally barely documented which makes evolution a time-consuming and risky process.

The communication between the programs and the database can be complex; many systems use dynamic SQL queries, according to which the SQL statements are built at runtime and sent to the database server through specific APIs. Moreover, increasingly popular object-relational mapping technologies allow programmers to communicate with the database by manipulating program objects, instead of writing SQL queries. This dynamicity makes difficult the process of evolution. In addition, heterogeneous systems, i.e., using several technologies to access its database, further complicate the maintenance task and require programmers to master several technologies. Therefore, DISS evolution clearly calls for automated support.

The main goal of this thesis is to invent and evaluate novel and efficient automated methods to support the evolution of dynamic and heterogeneous DISS. More particularly, the thesis aims at proposing methodologies, techniques and tools for (a) analyzing and understanding how the database and the application source code have (co-)evolved over time, in order to facilitate future developments; (b) supporting the adaptation of the application source code under database change.

ACKNOWLEDGEMENTS

Being a PhD student was an enriching experience. During this adventure, numerous people helped me accomplish this work, including colleagues, friends and family. I wish to thank all those people without whom I would have never completed this thesis.

First of all, I would like to heartily thank my supervisor, Prof. Anthony Cleve, who gave me the opportunity to perform this PhD research. I truly appreciated his wise advices, feedback and encouragements all over my thesis. I also enjoyed all those funny moments that we had together. A friend of mine once told me this sentence that I will never forget: *"Nous sommes restés dignes"*. Now, I can finally answer him: yes, we did.

Secondly, I would like to warmly thank the other members of my PhD committee, Prof. Michele Lanza, Prof. Tom Mens, Prof. Vincent Englebert, Prof. Benoit Frenay and Prof. Wim Vanhoof for their valuable remarks and time they spent reading the manuscript.

I also thank all my colleagues of the Faculty for this pleasing working atmosphere. Special thanks to my office colleagues, Csaba, Javier, Maxime, Nesrine and Adrien for their kindness. I am also grateful to Julie to have helped me conduct the controlled experiment with students. Another special thanks to Fabian for his friendship, advices and questions about SQL.

I would like to thank all my close friends for their encouragements and their enthusiasm. Special thanks to Gilles, Kevin, Sylvain, Tich and Warich.

Last but not least, I would like to thank my wonderful family, without whom none of this would have been possible. I thank my parents, Léo and Violaine for their support and love.

This PhD thesis was carried out in the context of FRFC PDR research project T.0022.13 *"Data-Intensive Software System Evolution"*, in collaboration with the University of Mons, and financed by the F.R.S.-FNRS Fonds de la Recherche Scientifique.

Contents

List of Algorithms

List of Figures

List of Tables

List of Listings

INTRODUCTION

Data-Intensive Software Systems

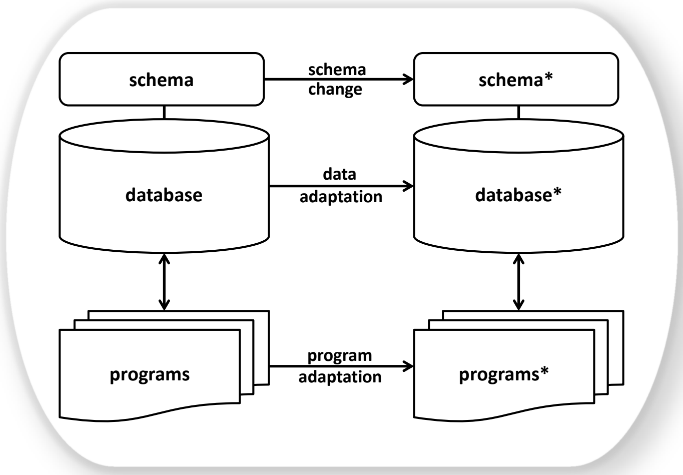
Nowadays, information systems represent important and crucial assets in most enterprises and organizations, since they often support the majority of their business activities, i.e., sale, production and management. Those systems are usually large software systems that manipulate a large amount of data, hence the name *data-intensive software system* (DISS). A DISS is generally composed of a software system and a database that must co-exist. The software system is composed of a collection of application programs which intensively interact with the database. The business data are stored in the database. Database experts perceive the database as the system's central component, around which the application programs are built. The database is possibly shared by many programs that simultaneously read, create, update and delete the stored data.

The Database Component

The goal of the database is to collect all the relevant data about the application domain of the system. During the database design process, several database schemas are produced. A database schema represents a model, i.e., an abstract formal representation of a given application domain. There exist different types of database schemas, belonging to different levels of abstraction. The first schema produced during the process, i.e., the conceptual schema, is the most abstract: it identifies and describes the domain entities, their properties and their associations in a platform-independent way. The last schema, i.e., the physical schema, is the most concrete schema and exactly specifies the actual database structures (collections, fields, constraints, ...) in a platform-specific way. The physical schema is then coded in the Data Description Language (DDL) of the database management system (DBMS).

Data-Intensive Software System Evolution

DISS are subject to continuous modification due to changes in the environment in which they operate. DISS evolution constitutes a complex process but is indispensable to keep systems adapted to ever-changing business needs and technological platforms. Both the software and database engineering research communities have addressed the problems of system evolution. Surprisingly, they have conducted



Data-intensive software system evolution.

very little research at the intersection of these two fields, where software meets data. Any change in the business requirements necessitates the synchronized adaptation of several components, such as the database schema, the database contents and the programs that interact with the database. Database schemas and application source code are supposed to be fully documented in order to make maintenance and evolution easier. Unfortunately, developers seldom have time to write and maintain a precise, complete and up to date documentation.

Due to this lack of documentation, this synchronized adaptation between the three components (database schema, stored data and application source code) is a complex, time-consuming and risky process. Indeed, in case of database schema change, programmers have to adapt the *outdated* queries - that still manipulate the older structures - to fit with the updated schema. If they do not properly handle those outdated queries, it can lead to program errors/crashes when the application will try to query non-existing structures at runtime. To illustrate the complexity of this adaptation process, Qiu *et al.* [Qiu et al., 2013] conducted an empirical analysis of the co-evolution of database schemas and code in ten popular large open-source data-intensive systems; their study revealed that, for each database revision containing 2-5 atomic schema changes, around 100 - 1,000 lines of code were changed by developers, which represents quite significant changes.

The Complexity of Data-Intensive Software Systems

The architecture of DISS is typically complex; the database schema can define hundreds/thousands of collections and fields and the implemented database can store millions of records, that can represent terabytes/petabytes of data (as illustration, Facebook data warehouse stores hundreds of petabytes[1]). Those data are shared and manipulated by different application programs through a given data manipulation language (DML), e.g., SQL language, COBOL DML, DL/I.

The application programs are written in their programming languages and can represent millions of lines of code. The communication between the programs and the database can be complex as well; many systems use *dynamic SQL* queries, according to which the SQL statements are built at runtime and sent to the database server through specific APIs. Moreover, increasingly popular object-relational mapping (ORM) technologies allow programmers to communicate with the database by manipulating program objects - instead of writing SQL queries - without having to know how those objects relate to their data sources.

Adapting the application source code to any database schema change necessitates to (1) precisely spot the code locations to modify (i.e., accessing the older database structures) and (2) understand how to adapt those locations to the new structures. Therefore, since database accesses are increasingly *dynamic* and popular technologies like ORMs allow programmers to use an external object-oriented schema of the database to access it (thus, both physical and ORM schemas should evolve synchronously over time), the process of program-database co-evolution becomes complex, time-consuming and error-prone. In addition, *heterogeneous* systems, i.e., using several technologies to access its database, further complicate the maintenance task and require programmers to master several technologies.

Goal of the Dissertation

The main goal of this dissertation is to devise and evaluate novel and efficient automated methods to support the evolution of dynamic and heterogeneous DISS. More particularly, the dissertation aims at proposing methodologies, techniques and tools for :

(a) *analyzing* and *understanding* how the database schema and the application source code have (co-)evolved over time, in order to facilitate future developments.
(b) *supporting* the adaptation of the application source code under database schema evolution.

We formulate our thesis as:

> *Analyzing the evolution history of dynamic and heterogeneous data-intensive software systems and extracting database accesses from source code, are useful to support their evolution.*

[1] http://bit.ly/1hyvJDn

Research Questions

This thesis research is driven by the following research questions (RQ):

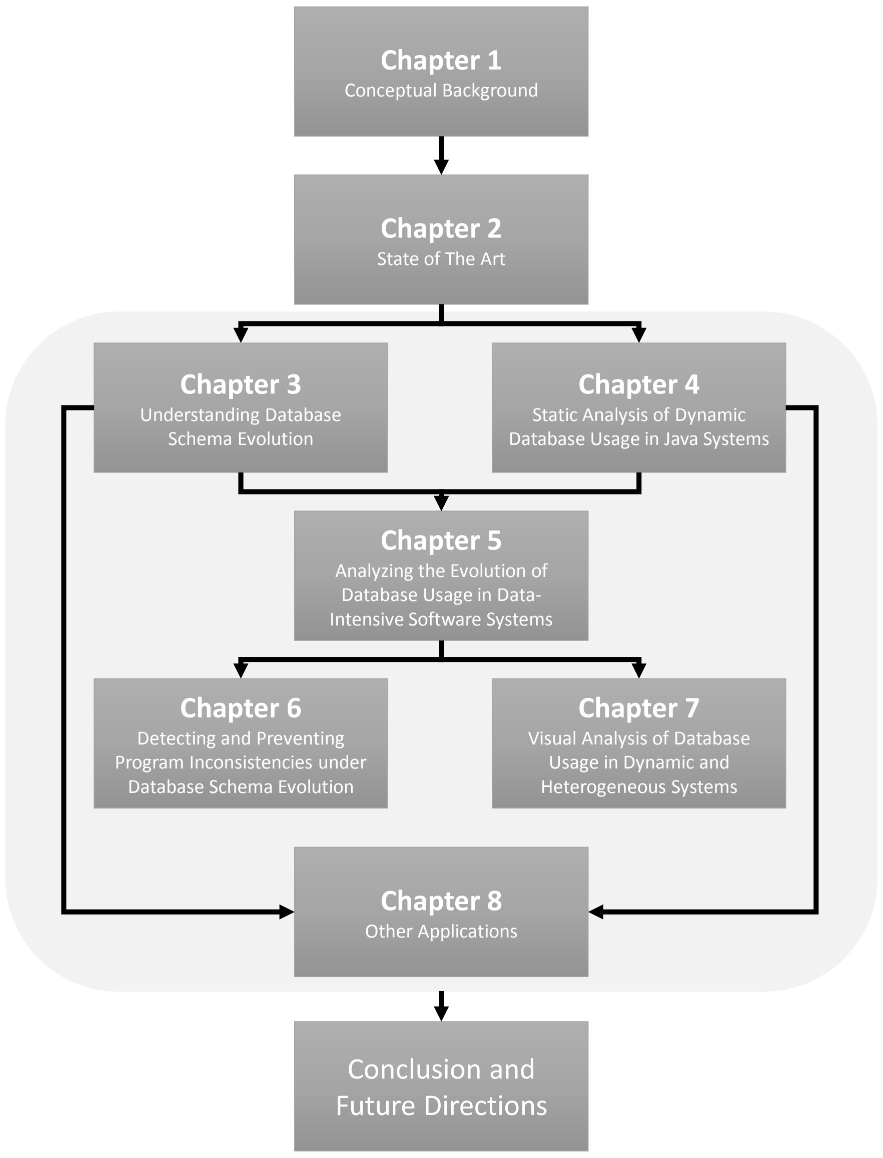
- **RQ1:** How can history analysis of a DISS support the actual maintenance of the system?

- **RQ2:** How to automatically analyze and extract the communication between application programs and the database in a *dynamic* DISS?

- **RQ3:** To what extent can we support program-database co-evolution in *dynamic* and *heterogeneous* DISS?

Dissertation Outline

The remainder of this dissertation is composed of 8 chapters, as depicted below.


Chapter 1
Conceptual Background
Chapter 2
State of The Art
Chapter 3
Understanding Database Schema Evolution
Chapter 4
Static Analysis of Dynamic Database Usage in Java Systems
Chapter 5
Analyzing the Evolution of Database Usage in Data-Intensive Software Systems
Chapter 6
Detecting and Preventing Program Inconsistencies under Database Schema Evolution
Chapter 7
Visual Analysis of Database Usage in Dynamic and Heterogeneous Systems
Chapter 8
Other Applications
Conclusion and Future Directions

General structure of the dissertation.

Chapter 1 provides a brief introduction to the main basic concepts used in the thesis.

Chapter 2 presents the state-of-the-art in the field of data-intensive software system evolution and outlines some gaps in the literature. It finally recapitulates the problems we are addressing in this thesis.

Chapter 3 explores the use of the database schema evolution history as an additional information source to aid database reverse engineering. It presents a tool-supported method for analyzing the evolution history of legacy databases, and reports on a large scale case study of reverse engineering a complex information system.

Chapter 4 presents an analysis approach that aims to statically detect database accesses within the source and to recover their actual SQL values. This automatic approach, specifically designed for Java systems, targets at three main database access technologies, i.e., JDBC, Hibernate and JPA. It secondly evaluates this approach on three real-life systems.

Chapter 5 builds on the static analysis approach presented in Chapter 4. It presents a historical analysis approach that allows us to analyze, at a fine-grained level, how the program source code and the database schema have co-evolved over time. It then motivates the benefits of this historical approach on real-life systems.

Chapter 6 presents a tool-supported approach that allows developers to 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. Based on the historical analysis presented in Chapter 5, this chapter proposes an approach that has been specifically designed to deal with Java systems that use dynamic data access frameworks such as JDBC, Hibernate and JPA. It finally motivates and evaluates the proposed approach, on three real-life systems of different size and nature.

Chapter 7 presents DAHLIA 2.0, a visualization tool that allows developers to analyze the database usage in dynamic and heterogeneous systems by visualizing the interactions between the application program and the database. It then applies DAHLIA 2.0 to real-life systems and illustrates the benefits of this visualization. Finally, it presents a controlled experiment for the empirical evaluation of DAHLIA 2.0. The objective of this experiment is to assess the suitability of our approach in the context of program-database co-evolution.

Chapter 8 presents three different applications of the approaches presented in Chapters 3 and 4, namely (1) concept location for DISS, (2) database reverse engineering and (3) database schema evolution in schema-less NoSQL data stores.

This dissertation ends with a summary and an evaluation of the thesis contributions with respect to our research questions. We also identify future research directions.

Related Publications

Most of the research results presented in this thesis have been published as book chapters, journal articles or international conference papers. Below, we list these publications:

(1) Meurice, L. and Cleve, A. (2014). DAHLIA: A visual analyzer of database schema evolution. In Proceedings of the CSMR/WCRE 2014 Software Evolution Week, pages 464–468. IEEE Computer Society, 2014. [Meurice and Cleve, 2014]

(2) Meurice, L., Ruiz, F., Weber, J., and Cleve, A. (2014). Establishing referential integrity in legacy information systems - Reality bites! In Proceedings of the 30th European Conference in Software Maintenance and Evolution (ICSME 2014), pages 461–465. IEEE Computer Society, 2014. [Meurice et al., 2014]

(3) Cleve, A., Gobert, M., Meurice, L., Maes, J., and Weber, J (2015). Understanding database schema evolution: A case study. In Science of Computer Programming, pages 113–121. Elsevier, 2015. [Cleve et al., 2015]

(4) Nagy, C.,Meurice, L., and Cleve, A. (2015). Where was this SQL query executed? a static concept location approach. In Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pages 580–584. IEEE Computer Society Press, 2015. [Nagy et al., 2015]

(5) Meurice, L., Goeminne, M., Mens, T., Nagy, C., Decan, A., and Cleve, A. (2016). Analysing the Evolution of Database Usage in Data-Intensive Software Systems. In Software Technology : 10 Years of Innovation in IEEE Computer, John Wiley & Sons/IEEE Computer Society Press, 2016 (to appear). [Meurice et al., 2016a]

(6) Meurice, L., Nagy, C., and Cleve, A. (2016). Static analysis of dynamic database usage in java systems. In Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAISE 2016), pages 491–506. Springer Verlag, 2016. [Meurice et al., 2016c]

(7) Meurice, L., Nagy, C., and Cleve, A. (2016). Detecting and preventing program inconsistencies under database schema evolution. In Proceedings of the International Conference on Software Quality, Reliability and Security.(QRS 2016), pages 262-273. IEEE Computer Society Press, 2016. **Best Paper Award**. [Meurice et al., 2016b]

(8) Meurice, L. and Cleve, A. (2016). DAHLIA 2.0: A Visual Analyzer of Database Usage in Dynamic and Heterogeneous Systems. In Proceedings of the 4th IEEE Working Conference on Software Visualization (VISSOFT 2016), pages 76-80. IEEE Computer Society Press, 2016. [Meurice and Cleve, 2016]

(9) Meurice, L. and Cleve, A. (2017). Supporting Schema Evolution in Schema-less NoSQL Data Stores. In Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017), pages 457-461. IEEE Computer Society Press, 2017. [Meurice and Cleve, 2017]

CHAPTER 1

Conceptual Background

This chapter provides the reader with a conceptual background, by introducing the main concepts that we will use in this thesis: database engineering, database reverse engineering, program analysis, the object-relational mapping technologies and the use of popular Java database access technologies.

1.1 Database Engineering

Database engineering is a process comprising a series of phases, such as database design, implementation and maintenance. This process relies on (or let say, *should* rely on) a disciplined approach divided into several phases. Figure 1.1 gives an abstract representation of the database engineering process. During this process different database schemas are produced. A database schema represents a model, i.e., an abstract formal representation of a given application domain. Such a model allows to better understand this application domain and to build an operational database allowing to store and manipulate information about it. There exist different types of database schemas, belonging to different levels of abstraction. The first schema produced during the process is the most abstract (platform-independent) while the last one is the most concrete (platform-specific) and exactly specifies the actual database structures.

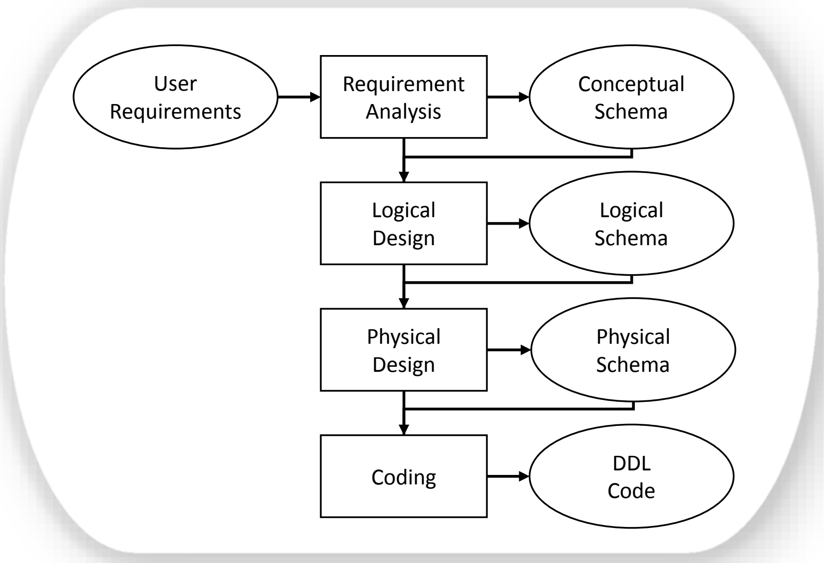


Figure 1.1: Database engineering process.

Requirement analysis. The *requirements analysis* phase, also known as conceptual analysis, consists in collecting the user requirements about the application domain under consideration. With this requirements collection, the database engineer will produce the conceptual schema of the future database. A conceptual schema extracts pertinent concepts of the domain of application from the user requirements, which are typically expressed in natural language.

Logical design. The second phase, called *logical design*, consists in extracting the logical schema of the future database from the conceptual schema obtained so far. This logical schema describes more concretely the database structures, the relationships between these structures and the integrity constraints. It complies with a given logical model that is compatible with a given database paradigm.

Physical design. The *physical design* phase consists in designing a physical schema. A physical schema is obtained by enriching the logical schema with technical constructs needed to make the future database efficient and robust (e.g., indexes and physical dbspaces).

Coding phase. Finally the *coding phase* consists in translating the physical schema into the executable database code. The generated code, also called data definition language (DDL) code, will be compatible with the particular database management system (DBMS) and will allow defining and creating the database structures.

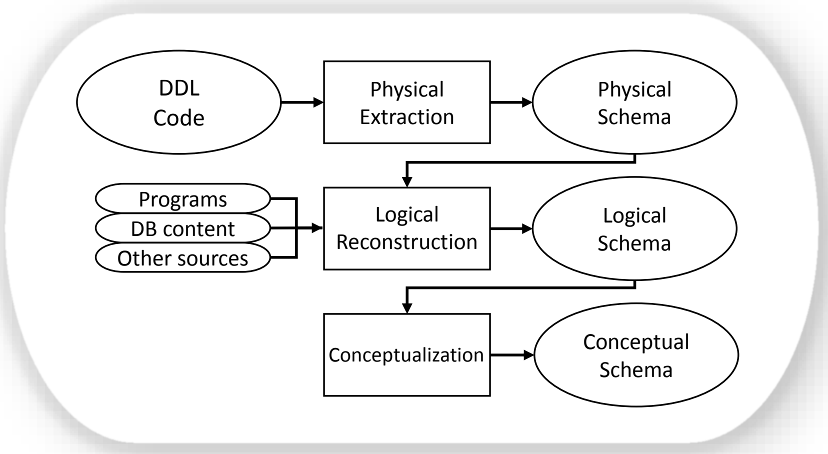


Figure 1.2: Database reverse engineering process.

1.2 Database Reverse Engineering

During its lifetime, a database is usually faced with evolution; the developers often have to make changes to their database structures and constraints in order to adapt the database to ever-changing needs. However, such changes are not always trivial and can represent a complex task. Indeed, many existing (legacy) databases have not been designed in a disciplined way and sometimes, no systematic database engineering process was followed during the design phase. Therefore, it is common to encounter databases that are poorly/not documented. Even worse, it is frequent that the DDL code actually constitutes the only available documentation of the database. The process of recovering missing database schemas, and in particular of the conceptual schema, is called **database reverse engineering**. Database reverse engineering (DRE) is globally considered as the extraction of conceptual schema from several information sources [Hainaut, 2002]. This process is illustrated by Figure 1.2. The database reverse engineering process is composed of three successive steps.

Physical extraction. This phase aims to produce the physical schema from the DDL code. The physical extraction produces a *raw* physical schema, that contains all the explicit constructs expressed in the DDL code but does not include potentially implicit database constructs like, for example, undeclared foreign keys.

Logical reconstruction. This phase aims to produce a *refined* logical schema from the physical schema by analyzing additional artifacts. For instance, implicit schema constructs can be recovered by analyzing information sources such as programs source code and database contents.

Conceptualization. This final step produces the database conceptual schema from the refined logical schema.

1.3 Program Analysis

Program comprehension is a crucial process in software maintenance/evolution that is necessary in order to sufficiently understand the program before its modification. This activity has received the attention from the research community, particularly over the last decade. Program understanding requires the analysis of such artefacts as source code and documentation but it is not sufficient to get a complete view of the program/system. Indeed, analyzing the program behaviour at runtime can provide additional information pertaining to what the program is doing in specific execution scenarios.

1.3.1 Static Analysis

A natural way to proceed in the program comprehension process is by means of *static program analysis* techniques. This kind of analysis consists in analyzing the source code of the program in order to extract static information and derive program properties of interest. Then, by studying the extracted properties, one can obtain a better understanding of the program. Static analysis provides a better comprehension of the structure, the dependencies and the behaviour of the program but without executing it. Several static analysis techniques exist such as:

- Control-flow analysis [Allen, 1970]: the control-flow analysis is a static analysis technique for determining the control flow of a program. The control flow is expressed as a control flow graph, namely a graph representation of all paths that might be visited through a program during its execution.
- Data-flow analysis [Kildall, 1973]: data-flow analysis is a static analysis technique designed to gather information about the values at each point of the program and how they change over time. The control flow graph is used to determine which particular value is assigned to a particular variable. Moreover, two categories of data-flow analysis emerge, i.e., the *intraprocedural* and *interprocedural* data-flow analyses.
  a) Intraprocedural analysis: the intraprocedural data-flow analysis operates on a control-flow graph for a single method.
  b) Interprocedural analysis: the interprocedural analysis uses calling relationships among procedures. It analyzes the effects of (1) procedure calls in the caller procedures, and (2) calling contexts in the callee procedures. The analysis can be performed by exploiting the program *call graph*, i.e, a control flow graph which represents calling relationships between procedures; each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g.
- Model checking [Emerson and Clarke, 1980; Queille and Sifakis, 1982]: model checking refers to the problem to automatically check if a given model complies with the given specification. The idea is to determine if a correctness

property is verified by exhaustively exploring the reachable states of a program. If the property does not hold, the model checking algorithm generates an execution trace leading to a state in which the property is violated.

- Symbolic execution [King, 1976]: symbolic execution is a static analysis technique to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs instead of actual inputs coming from a normal program execution.
- Static program slicing [Weiser, 1981]: program slicing is the computation of the set of programs statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criterion. As illustration: given a program P and a slicing criterion $C = (i, V)$ where i is a statement in program P and V is a subset of variables in P, a static program slice consists of all statements in program P that may affect variable v for a set of all possible inputs at the statement i.

1.3.2 Dynamic Analysis

In contrast, *dynamic program analysis* techniques focus on information generated at runtime. This second category of program analysis techniques consists in analyzing the properties of the running program, i.e., at execution time. Dynamic analysis allows mining a precise (but partial) picture of the program during its execution. On the one hand it can be more accurate than static program analysis, but on the other hand, it is obviously restricted to some execution paths of the program.

In summary, dynamic program analysis typically aims to analyze information built at runtime while static analysis handles information extracted from source code. However, one can notice that both techniques are complementary for achieving a complete program understanding process.

1.4 Object-Relational Mapping

Object-relational mapping (ORM) is a programming technique used to connect object code to a relational database. Object code is written in object-oriented (OO) programming languages. ORM converts data between type systems that are unable to coexist within relational databases and OO programming languages. ORM offers mechanisms to define mappings between database objects (e.g., tables and columns) and source code objects (e.g., classes and fields/attributes). It thus allows developers to manipulate source code objects without having to consider how those objects relate to their data sources. In addition, ORM generally uses a SQL database driver to translate those manipulations into corresponding SQL accesses. Figure 1.3 represents a database architecture in presence of an ORM.

ORM mainly serves to tackle the so-called object-relational impedance mismatch. The object-relational impedance mismatch [JBOSS Hibernate, 2017] is a set of conceptual and technical difficulties that are often encountered when a relational DBMS is being served by an application program written in an object-oriented programming language (e.g., inheritance, aggregation, polymorphism, ...).

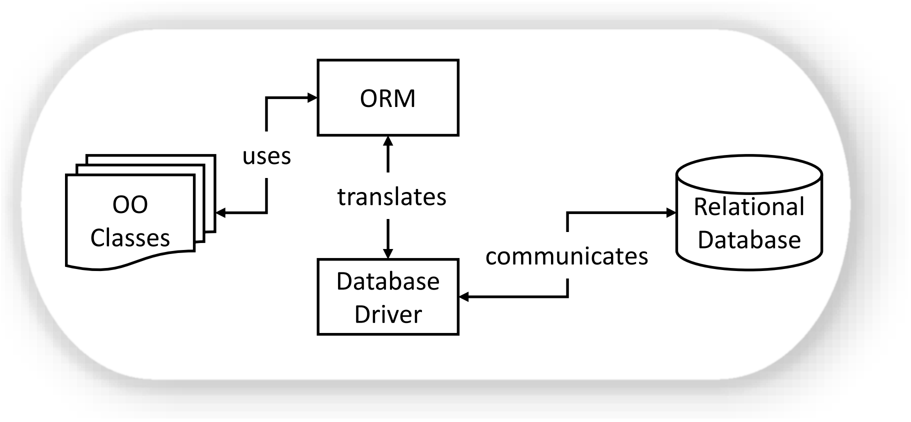


Figure 1.3: Database architecture in presence of an ORM.

1.5 Java and its Database Access Technologies

To facilitate the data management and manipulation in DISS, a large variety of database access technologies are proposed and used, especially for systems developed in popular languages such as Java. Whereas Java is the most popular programming language today [TIOBE Programming Community Index, 2017], Goeminne et al. [Goeminne and Mens, 2015] carried out a large-scale empirical study and revealed that three particular Java database access technologies were emerging, i.e., JDBC, Hibernate and JPA. Below we briefly introduce JDBC, Hibernate and JPA, by illustrating their underlying database access mechanisms.

1.5.1 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 Listing 1.1).

1.5.2 Hibernate

Hibernate is an 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). The mapping is usually defined by use of Hibernate configuration files (i.e., `.hbm.xml`). Listing 1.2 shows an example of Hibernate mapping defined between the `customer` table and the `Customer` class.

Hibernate provides also a SQL-inspired language called *Hibernate Query Language* (HQL) which allows to write SQL-like queries using the mappings defined before. Listing 1.3 provides an example of a HQL query execution (line 13). In

```
 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.executeQuery(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   }
21 }
```

Listing 1.1: Java code fragment using the JDBC API to execute a SQL query (line 10).

```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-mapping>
  <class name="Customer" table="customer">
    <id name="customer_id" column="id" type="integer" />
    <property name="name" column = "name" not-null="false" type="string" length="255" />
    <property name="city" column="city" not-null="false" type="string" length="255" />
    ...
  </class>
</hibernate-mapping>
```

Listing 1.2: Example of mapping defined in a Hibernate configuration file. The `customer` table and its columns are mapped to the `Customer` class and its attributes.

addition, *Criteria Queries* are provided as an object-oriented alternative to HQL, where one can construct a query by simple method invocations. See Listing 1.4 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. Listing 1.5 illustrates a sample record insertion in the database.

1.5.3 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. JPA uses Java annotations to establish this mapping. Listing 1.6 illustrates an example of mapping between the `customer` table and the `Customer` class by use of JPA annotations.

JPA offers 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

```
1  public class ProductDaoImpl implements ProductDao {
2  
3    private SessionFactory sessionFactory;
4  
5    public void setSessionFactory(SessionFactort 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   }
15 }
```

Listing 1.3: Java code executing a HQL query (line 13). Product selection according to its category.

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

Listing 1.4: Java code executing a Criteria query. Customer selection restricted on the name and city.

```
private Session session;
...
public void saveCustomer(Customer myCustomer) {
  saveObject(myCustomer);
}

public void saveObject(Object o) {
  session.save(o);
}
```

Listing 1.5: Hibernate operation on a mapped entity class instance. Insertion of a new customer.

```
@Entity
@Table(name = "customer")
public class Customer{
  @Id
  @Column(name = "customer_id")
  protected int id;

  @Column(name = "name")
  protected String name;

  @Column(name = "city")
  protected String city;

  ...
}
```

Listing 1.6: Example of mapping defined by use of JPA annotations. The `customer` table and its columns are mapped to the `Customer` class and its attributes.

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

Listing 1.7: Sample JPQL query. Customer selection according to a given id.

```
1 EntityManagerFactory emf = ...;
2 EntityManager em = emf.createEntityManager();
3 em.getTransaction().begin();
4 Order order = ...;
5 em.persist(order);
6 em.getTransaction().commit();
7 em.close();
```

Listing 1.8: JPA operation on a mapped entity class instance. Creation and insertion of a new order (line 5).

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. Listing 1.7 shows an example of JPQL query execution. JPA also provides a way to perform CRUD operations on the instances of mapped entity classes. For instance, Listing 1.8 illustrates the creation and insertion of a new order in the database.

1.5.4 Dynamically generated Queries

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 generated at runtime, through String concatenations. In Listing 1.1, the value of the SQL query executed at line 10 depends on the path followed by the program at runtime. The SQL query value is stored in variable *sql*; *sql* is initialized at line 7 but its value can change according to the followed program path. For instance, its value will be updated if the program passes through the boolean condition at line 8. Moreover, line 9 utilizes *y*, an input parameter of the method, to update the value of *sql*.

Furthermore, we observe that SQL queries are not always written in the programs, but generated in the background. That is the case when the application accesses its database by use of an ORM framework (e.g., Hibernate or JPA). As illustration, line 4 of Listing 1.5 saves a new customer in the database. In this example, no SQL queries are written in the source code; however, the program will send the corresponding SQL access to the database server at runtime - i.e., a SQL query on the following form: `INSERT INTO customer(customer_id, name, city, ...)  VALUES(...)`.

Roadmap

This chapter has briefly presented the conceptual background of the thesis. In the next chapter (Chapter 2), we summarize the state-of-the-art in the field of data-intensive software system evolution and outline some gaps in the literature.

CHAPTER 2

STATE OF THE ART

In this chapter, we introduce the historical context of software engineering, software evolution and more particularly, data-intensive software system evolution. We present the state-of-the-art in this field and outline some gaps in the literature. We finally recapitulate the problems we are addressing in this thesis.

2.1 Software Engineering: the Origins

The origins of the term "Software Engineering" were used for the very first time in 1968 as a title for the NATO conference on Software Engineering [Naur and Randell, 1969]. This international conference was attended by experts, from different countries, all coming from professional domains concerned with software. This conference was organized in a period of crisis in the software development industry (poor quality of software, projects exceeding time and budget, ...). The main goal of this conference was to define best practices in order to obtain reliable, efficient and economically viable software.

In 1970, Royce described his personal views about managing large software developments [Royce, 1987]. Based on his experience, he realized that there were two essential steps common to all computer program developments, regardless of size or complexity, namely the *analysis* step followed by the *coding* step. For small software development projects these two steps were sufficient, but not for the development of larger software systems. These require many additional steps back and forth, which give development an iterative nature. As consequence, Royce defined the well-known *waterfall life-cycle* process for software development. In this model, the *maintenance* phase is the final phase of the life-cycle of a software system, after its deployment. A common perception of maintenance is that it merely involves bug fixes and minor adjustments to the software. Later, software maintenance became a part of the IEEE Standard for Software Maintenance [IEEE, 1999] and was defined as "the modification of a software product *after* delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment".

2.2 Software Evolution: Towards a new Research Area

In the seventies, Lehman demonstrated that systems continue to evolve over time. He observed that, when evolving, the systems grow more complex. His research led to his famous laws of *software evolution* [Lehman, 1980, 1984, 1996; Lehman et al., 1997]. This was probably the first time that the term "software evolution" was evoked. Lehman emphasized the difference between software evolution and the *after deployment* side of software maintenance. Research on software evolution has become popular due to those laws. This has inspired many researchers to validate/revisit these laws on open source software systems [Godfrey and Tu, 2000; Fernández-Ramil et al., 2008; Wermelinger et al., 2011; Skoulis et al., 2014].

Nowadays, software evolution has become a very popular and active research area in software engineering, and the terms software evolution and software maintenance are often used as synonyms. The international ISO/IEC 14764 standard for software maintenance [International Standards Organisation (ISO), 1999], stresses the importance of pre-delivery activities as well as the post-delivery activities of maintenance. Four categories of software maintenance are catalogued:

(a) *Corrective* maintenance: identifying and fixing errors present in a software product. It consists in correcting discovered problems.
(b) *Adaptive* maintenance: modifying the software in order to satisfy a changing environment.
(c) *Perfective* maintenance: modifying the software product to satisfy new user requirements.
(d) *Preventive* maintenance: modifying the software product to increase maintainability and reliability. This category of maintenance aims to prevent eventual problems in the future.

Nowadays, one considers that software maintenance represents on average about 60 percent of the entire software development costs and it is therefore the most costly phase of the software life cycle [Glass, 2001].

Many renowned researchers have contributed to this research domain, working on a wide variety of topics such as test and production code [Zaidman et al., 2011], code cloning [Göde and Koschke, 2011], bug prediction [Guo et al., 2010; D'Ambros et al., 2012], build systems [McIntosh et al., 2012], change-proneness and fault-proneness [Khomh et al., 2012], and many more. Another emerging trend in contemporary empirical research is the focus on the social aspects, focusing on the people that are involved in the software development process as opposed to the software artefacts themselves [Nakakoji et al., 2002; Weiss et al., 2006; Ye et al., 2005;

Robles et al., 2009; Devanbu et al., 2009; Antwerp and Madey, 2010; Canfora et al., 2011]. With respect to measuring the quality of software [Chidamber and Kemerer, 1994], especially in the context of software evolution, many research advances were performed [Lanza et al., 2005; Pfleeger, 2008; Darcy et al., 2010; Vasilescu et al., 2011; Mordal-Manet et al., 2013].

2.3 Data-Intensive Software System Evolution

In a continuously evolving environment, system evolution is a complex task. This process is even more complicated in the case of data-intensive software systems. Indeed, those systems tend to access their underlying database in an increasingly dynamic way; maintaining the executed queries is a complicated task since they are usually constructed and generated at runtime by the program. Moreover, popular access technologies like ORMs are emerging; since programmers use an external object-oriented schema of the database, they have to synchronously evolve the object-oriented schema and the database schema.

Both the software and database engineering research communities have addressed the problem of system evolution. However, very little research has been conducted at the junction of these two fields.

2.3.1 Database Reverse Engineering

Data-intensive software system evolution often relies on the availability of up-to-date database documentation. However, in practice, the documentation may be incomplete, obsolete or simply missing. As illustration, the conceptual, logical and physical database schemas are often needed to ensure the evolution task. However, sometimes the DDL code constitutes the only available database documentation. Furthermore, this code is often incomplete since some data structures and constraints cannot be expressed in DDL. Therefore, the process of recovering those implicit properties may prove indispensable but requires additional effort.

Several research works have been conducted on that field and the proposed database understanding techniques exploit different information sources.

(a) Database schema analysis [Navathe and Awong, 1988; Markowitz and Makowsky, 1990; Premerlani and Blaha, 1994]. Analyzing the database schema structures may help identify hidden constructs such as relationships and hierarchies between data.

(b) Data analysis [Chiang et al., 1994; Lopes et al., 2002; Yao and Hamilton, 2008; Pannurat et al., 2010]. Mining the database contents can be used in two ways. Firstly, to discover implicit properties, such as functional dependencies and foreign keys. Secondly, to check hypothetic constructs that have been suggested by other means.

(c) Graphical User Interface (GUI) analysis [Terwilliger et al., 2006; Ramdoyal et al., 2010]. Forms, reports and dialog boxes are user-oriented views on the database that exhibit spatial structures, meaningful names, explicit usage guidelines and, at runtime, data population and error messages that can provide information on data structures and constraints.

(d) Static program analysis [Petit et al., 1994; Di Lucca et al., 2000; Cleve et al., 2006]. Analysis, such as data-flow graph exploration, can bring valuable information on field structure and meaningful names. More sophisticated techniques such as program slicing can be used to identify complex constraint checking.

(e) Dynamic program analysis [Grosso et al., 2007; Cleve and Hainaut, 2008; Alalfi et al., 2009; Cleve et al., 2012]. In the case of highly dynamic program-database interactions, the database queries may only exist at runtime. Hence recent techniques allow to capture and analyze SQL execution traces in order to retrieve structural information.

It is important to note that none of those sources of information is sufficient, yet they can all contribute to a better knowledge of the hidden components and properties of a database schema.

2.3.2 Database Schema Evolution Analysis

It has been already shown that analyzing the *system evolution history* can provide valuable data for *history-based recommenders*. For instance, Ying *et al.* [Ying et al., 2004] and Zimmermann *et al.* [Zimmermann et al., 2005] 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.

Moreover, Rahm *et al.* [Rahm and Bernstein, 2006] discussed the growing interest of schema evolution in recent research works; the authors built an online bibliography aiming to provide a comprehensive and up-to-date collection of publications on schema evolution. They categorized publications along multiple hierarchical dimensions; they did not limit themselves to database schema evolution but they also considered related fields such as ontology evolution, software evolution and workflow evolution.

Sjøberg [Sjøberg, 1993] showed, already in 1993, potential uses of the *database schema evolution history*. He studied the schema evolution history of a large-scale medical application and showed, by using a thesaurus tool, that even a small change to the schema may have major consequences for the rest of the application code. The study reveals that schema changes are significant both in the development period and after the system has become operational. The consequences of the schema changes on the application programs have been measured. In particular, the tool provides information about how many screens, actions, queries, etc. may be affected by a possible schema change. The results confirm that change management tools are needed.

Curino *et al.* [Curino et al., 2008] present a study of the structural evolution of the Wikipedia database, with the aim to extract both a micro-classification and a macro-classification of schema changes. They also study the frequency distribution of those schema changes. The authors propose, in addition to a schema evolution statistics extractor, a tool that operates on the differences between subsequent schema versions and semi-automatically extracts the set of possible schema changes that have been applied. In this study, a period of four years has been considered, corresponding to 171 successive versions of the Wikipedia database schema. The latter is rather limited in size: it includes from 17 to 34 tables depending on the schema version considered. The total number of columns in the schema does not exceed 250, whatever the version. Their study shows an urgent needs for better automation and documentation tools for supporting graceful schema evolution.

Vassiliadis *et al.* [Vassiliadis et al., 2015, 2017] studied the evolution of individual database tables over time in eight different software systems. They report on their observations on how evolution-related properties, like the possibility of deletion, or the amount of updates that a table undergoes, are related to observable table properties like the number of attributes or the time of birth of a table.
Through a large-scale study on the evolution of database, they also tried to determine whether Lehman's laws of software evolution hold for evolving database schemas as well [Skoulis et al., 2014]. They conclude that the essence of Lehman's laws remains valid in this context, but that specific mechanics significantly differ when it comes to schema evolution.

Finally, recent approaches and studies have focused on the evolution of NoSQL databases. In [Scherzinger et al., 2015], the authors present ControVol, a framework controlling schema evolution in NoSQL applications. ControVol statically type checks object mapper class declarations against earlier versions in the code repository. ControVol is capable of warning developers of risky cases of mismatched data and schema. ControVol also suggests and performs automatic fixes to resolve possible schema migration problems.
Scherzinger *et al.* [Scherzinger et al., 2013] present a model checking approach to reveal scalability bottlenecks in NoSQL schemas.
Ringlstetter *et al.* [Ringlstetter et al., 2016] analyzed how developers evolve NoSQL document stores by means of evolution annotations. They discovered that evolution annotations are actually used; however, developers do not employ them for evolving the data model but for other tasks.

Discussion

While little research has focused on database schema evolution, we observe the interest of several authors for this field. Table 2.1 presents a brief summary of this related research. The first column contains the related work references. The second and third columns show the importance of the studied system(s) in terms of, respectively, database schema size and time period of analysis. In case of multiple studied systems, we retain the most important figures. The fourth column categorizes the presented approach; either an approach intended for comprehending the evolution

WORK	DB SIZE	PERIOD	APPROACH	GOAL
[SJØBERG, 1993]	666 fields	1 year	single-system	measuring modifications to database schemata and their consequences
[CURINO ET AL., 2008]	34 tables, 242 fields	4.5 years	single-system	extracting both a micro-classification and a macro-classification of schema changes
[VASSILIADIS ET AL., 2015]	≤858 fields, ≤114 tables	≤13 years	empirical	studying the evolution of individual database tables
[SCHERZINGER ET AL., 2015]	13 fields	2 years	single-system	framework controlling schema evolution in NoSQL applications using object mappers
[RINGLSTETTER ET AL., 2016]	-	-	empirical	analyzing how developers evolve NoSQL document stores by means of evolution annotations

Table 2.1: Summary of research works focusing on database schema evolution.

of a particular system (i.e., *single-system* approach), or an *empirical* study aiming at detecting general evolution trends and phenomenons. Finally, the fifth column briefly reminds the main objective of each work.

Some interesting observations can be made after analyzing the related work.

- Fewer studies propose to analyze the evolution history of database schemas to facilitate the future developments. Instead, the presented approaches generally target at the history analysis of one or multiple systems with the objective to detect general evolution trends.

- The presented works generally define and use their approach on small or medium database schemas (in terms of tables and columns) and periods.

- None of the presented approaches focus on analyzing database schema evolution in order to provide developers with recommendations about *what* to change in case of schema modifications and *who* the most appropriate person is for achieving a particular database-related activity.

2.3.3 Database Usage Analysis

Several researchers have tried to identify, extract and analyze database *usage* in application programs. The purpose of these approaches ranges from error checking [Christensen et al., 2003; Gould et al., 2004; Wassermann et al., 2007], SQL injection vulnerability detection [Wei et al., 2006; Fu et al., 2007], SQL fault localization [Clark et al., 2011], fault diagnosis [Javid and Embury, 2012] to impact analysis for database schema changes [Maule et al., 2008; Wang et al., 2010].

A pioneer work was published by Christensen *et al.* [Christensen et al., 2003], 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 present several applications of this analysis, such as statically checking the syntax of dynamically generated expressions, such as SQL queries. They evaluate their approach on Java classes with at most 4 kLOC.

Gould *et al.* propose a static analysis technique close to a pointer analysis, based on an interprocedural data-flow analysis to verify the correctness of dynamically generated query strings [Gould et al., 2004; Wassermann et al., 2007]. They also detail a prototype tool based on the algorithm and present several illustrative defects found in small size subject systems.

Wei *et al.* propose a technique to defend against the SQL injection attacks targeted at stored procedures. Their technique combines static analysis with runtime validation to eliminate the occurrence of such attacks. The static part consists of a stored procedure parser which detects and analyzes SQL statements potentially vulnerable to SQL injection attacks [Wei et al., 2006].

Fu *et al.* present SAFELI, a static analysis framework, designed for identifying SQL injection attack vulnerabilities at compile time. SAFELI statically inspects bytecode, using a symbolic execution, and finds out user inputs that could lead to security breaches [Fu et al., 2007].

Clark *et al.* presents a database-aware fault localization technique. Their technique uses SQL commands executed by the test suite to compute sets of statement-SQL and statement-attribute tuples. The technique computes suspiciousness scores for the statement-SQL tuples, statement-attribute tuples, and statements, then let the developer identify the database commands and program statements likely to cause failures [Clark et al., 2011].

Javid *et al.* focus on diagnosing failed test cases caused by embedded queries in database applications which are syntactically correct but semantically incorrect (i.e., they produce incomplete or incorrect results). They surveyed the literature and performed an experiment to evaluate the efficiency of existing techniques for this problem. They found that the existing techniques only provide a partial solution to this problem [Javid and Embury, 2012].

van den Brink *et al.* present a quality assessment approach for SQL statements embedded in PL/SQL, COBOL and Visual Basic code [Brink et al., 2007]. 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 [Ngo and Tan, 2008] 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.

More recently, Linares-Vasquez *et al.* [Linares-Vasquez et al., 2015] studied how developers *document* database usage in source code. Their results show that a large proportion of database-accessing methods is completely undocumented.
Later [Linares-Vásquez et al., 2016], they presented their tool approach, namely DBScribe, which aims at automatically generating always up-to-date natural language descriptions of database operations and schema constraints in source code methods. DBScribe statically analyzes the code and database schema to detect database usages and then propagates these usages and schema constraints through the call-chains implementing database-related features. Although DBScribe's implementation covers SQL-statements invoked by means of JDBC and Hibernate API calls, they do not manage Hibernate query language (HQL) and do not perform interprocedural analysis when resolving SQL-statements.

Chen *et al.* [Chen et al., 2014] 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. Their analysis revealed that the modifications made after analysis caused an important improvement of the studied systems' response time.
Later, the authors provide an experience report on (1) finding framework-specific database access bug patterns, (2) implementing a bug detection tool, and (3) integrating the tool into daily practice [Chen et al., 2016]. They discuss five database access bug patterns that they observed in large-scale industrial systems. Especially, they focus on bug patterns when using some frameworks/ORMs like Spring and Hibernate. Their objective is to help researchers create static bug detection tools which can be adopted in practice.

In [Goeminne and Mens, 2015], the authors carried out a coarse-grained historical analysis of the usage of Java relational database technologies (primarily JDBC, Hibernate, Spring, JPA and Vaadin) on several thousands of open source Java projects extracted from a GitHub corpus consisting of over 13K active projects [Allamanis and Sutton, 2013]. Using the statistical technique of survival analysis, they explored the survival of the database technologies in the considered projects. In particular, they analyzed whether certain technologies co-occur frequently, and whether some technologies get replaced over time by others. They observed that some combinations of database technologies appeared to complement and reinforce one another. They did not observe any evidence of technologies disappearing at the expense of others.

Discussion

Table 2.2 summarizes the main research related to database usage analysis. The first column contains the related work references. The second column categorizes the type of conducted database usage analysis. The third column indicates, in case of static analysis, if the approach proposes interprocedural analysis (i.e., which resolves values that are returned by interprocedural calls or values passed as arguments to the analyzed method/function). The fourth column indicates the targeted kind of database interactions. Finally, the fifth column reminds the main objective of each work.

Here again, some interesting observations can be made.

- Very few works propose automatic support for automatically detecting, extracting and accurately reconstructing the database usage of large systems. Most of the approaches only offer partial/time-consuming reconstructions of SQL statements manipulated within the source code.

- While only little research has been conducted on analyzing the database usage of systems using ORM technologies, almost none of the presented works cope with dynamic and heterogeneous systems, where several of those database access technologies co-exist (e.g., co-existing SQL and ORM-related statements).

2.3.4 Program-Database Co-Evolution

While the community shows a growing interest in database schema evolution [Rahm and Bernstein, 2006], researchers have only recently started to pay more attention to the analysis of the co-evolution of database schema and application code.

Lin *et al.* [Lin and Neamtiu, 2009] study the so-called *collateral* evolution of applications and databases, in which the evolution of an application is separated from the evolution of its persistent data, or from the database. They 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 observed that collateral evolution can lead to potential problems. However, the number of schema changes reported is very limited. In Mozilla, 20 table creations and 4 table deletions are reported in a period of 4 years. During 6 years of Monotone schema evolution, only 9 tables were added while 8 tables were deleted.

Qiu *et al.* [Qiu et al., 2013] conduct a large-scale empirical study on ten popular database applications from various domains to analyze how schemas and application code co-evolve. In particular, they study the evolution histories from the respective repositories to understand whether database schemas evolve frequently and significantly, how schemas evolve and impact the application code. In their approach, the authors try to estimate the impact of a database schema change in the code. This estimation is performed with a simple difference extractor calculating changed source lines between two versions.

WORK	ANALYSIS	INTER.	TECHNO.	GOAL
[CHRISTENSEN ET AL., 2003; GOULD ET AL., 2004; WASSERMANN ET AL., 2007]	static	✓	SQL	checking the validity of SQL queries
[WEI ET AL., 2006]	static + dynamic	✗	stored SQL procedure	preventing SQL injection vulnerability
[FU ET AL., 2007]	symbolic execution	✓	SQL	preventing SQL injection vulnerability
[CLARK ET AL., 2011]	dynamic		SQL	SQL fault localization
[JAVID AND EMBURY, 2012]	experiment		embedded SQL	fault diagnosis
[MAULE ET AL., 2008]	static	✓	SQL	impact analysis
[BRINK ET AL., 2007]	static	✗	embedded SQL	a quality assessment approach for SQL statements
[NGO AND TAN, 2008]	symbolic execution	✓	SQL	extracting database interactions from PHP source code
[LINARES-VASQUEZ ET AL., 2015]	interview		none	identifying how developers document database usages in source code
[LINARES-VÁSQUEZ ET AL., 2016]	static	✗	SQL	extracting database usage in Java source code to automatically redocument database-related methods
[CHEN ET AL., 2014]	static	✗	ORM	automatic framework to detect ORM performance anti-patterns
[GOEMINNE AND MENS, 2015]	study		SQL + ORM	analyzing the survival of database technologies

Table 2.2: Summary of research works focusing on database usage analysis.

Karahasanoić [Karahasanović, 2002] studied how the maintenance of application consistency can be supported by identifying and visualizing the impacts of changes in evolving object-oriented systems, including changes originating from a database schema. In particular, he focused on object-oriented databases rather than relational databases.

Goeminne *et al.* [Goeminne et al., 2014] 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). They empirically analyzed the evolution of the usage of SQL, Hibernate and JPA in a large and complex open source information system. Interestingly, they observed that the practice of using embedded SQL is still common today.

Using *what-if analysis* [Golfarelli et al., 2006] for changes that occur in the schema/structure of the database was proposed by Papastefanatos *et al.* [Papastefanatos et al., 2007, 2008, 2010]. 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.*

Maule *et al.* [Maule et al., 2008] propose static program analysis techniques for identifying the impact of relational database schema changes upon object-oriented applications. 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.* [Liu et al., 2013] propose a novel graph, called the attribute dependency graph, to show the dependencies between attributes in a database application and the programs involved. Their approach extracts the attribute dependency graph out of a database application from its source code by using a interprocedural static analysis. Their purpose was to aid maintenance processes, particularly impact analysis. They implemented their approach for PHP-based applications.

Gardikiotis and Malevris [Gardikiotis and Malevris, 2009] 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. They implemented their approach for PL/SQL applications.

Chang *et al.* [Chang et al., 2007] propose a formal framework for database refactoring which is based on a logical model of changes that can point to inconsistencies in the application code and modeling problems.

Curino *et al.* [Curino et al., 2008, 2009, 2013] present a tool called PRISM++ to help database administrators to predict and evaluate the effects on the applications due to changes to the database. PRISM++ includes an automatic schema evolution history analysis tool and a query rewriting engine as well to translate user queries across schema versions. However, PRISM++ relies only on the analysis of the database side.

Grolinger *et al.* [Grolinger and Capretz, 2011] examined the database schema evolution and proposed a unit test approach for the application code that accesses databases. Their objective is to evaluate the code against the altered database.

WORK	APPROACH	TECHNO	GOAL
[LIN AND NEAMTIU, 2009]	study	SQL	analyzing program-database co-evolution in open-source systems.
[QIU ET AL., 2013]	study	none	studying how programs and database schema co-evolve over time
[KARAHASANOVIĆ, 2002]	study	SQL	studying the impact of database schema changes in evolving OO systems
[GOEMINNE ET AL., 2014]	study	SQL + ORM	studying the co-evolution between code-related and database-related activities in DISS
[GOLFARELLI ET AL., 2006; PAPASTEFANATOS ET AL., 2007, 2008, 2010]	impact analysis	SQL	assessing impact of a database schema change over a set of queries and views provided by the user
[MAULE ET AL., 2008]	impact analysis	SQL	identifying the impact of relational database schema changes upon object-oriented applications
[LIU ET AL., 2013]	impact analysis	SQL	extracting the attribute dependency graph out of a database application
[GARDIKIOTIS AND MALEVRIS, 2009]	impact analysis	PL/SQL	identifying the source code statements affected by schema changes
[CHANG ET AL., 2007]	theoritical framework definition	none	proposing a formal framework for database refactoring
[CURINO ET AL., 2008, 2009, 2013]	database schema evolution automation	SQL	assisting developers in database schema evolution
[GROLINGER AND CAPRETZ, 2011]	unit test approach	SQL	proposing a unit test approach for the application code that accesses databases

Table 2.3: Summary of research works focusing on program-database co-evolution.

Discussion

Table 2.3 summarizes the presented related research. The first column contains the related work references. The second column categorizes the type of presented approach. The third column indicates the targeted kind of database access technologies. Finally, the fourth column reminds the main objective of each work.

Some interesting observations are made after analyzing the related work.

- Most of the related research works present studies analyzing how programs and database co-evolve over time. Very few works in the literature propose an impact analysis designed to assess the impact of a database schema change on the source code. Generally, the presented impact analysis approaches evaluate the impact of a database schema change on a set of queries/views/graphs provided by the user.

- In addition, one observes a lack of automatic support giving automatic *recommendations* to users about the modifications necessary to perform in the source code, at the line of code level, in case of database schema changes.

- Fewer approaches are designed to target heterogeneous systems, i.e., using several access technologies to access the database. More specifically, the presence of ORM technologies and their evolution with the source code are barely studied and supported, which constitutes an important challenge.

2.4 What kind of problems do we intend to address?

In this Chapter, we presented the main research tackling the problem of DISS evolution. In particular, we articulated this state-of-the-art around three main axes, namely database schema evolution, database usage analysis and program-database co-evolution.

For each axis, we made interesting observations concerning the existing techniques and approaches. The most striking observation is the severe lack of automatic support for DISS evolution.

(a) Focusing on the evolution history of the database schema with the aim to facilitate future developments is almost never considered by the literature. We claim that analyzing the database schema evolution history is valuable to understand the system and to facilitate DISS evolution.

(b) Detecting and analyzing the database usage in the source code often constitutes a prerequisite to DISS evolution. Moreover, the database accesses are often built in a more and more dynamic way in the code. Sometimes, the SQL query sent to the database server is completely hidden to the user because automatically produced by the ORM layer. In addition, several access technologies can coexist in a same DISS, which makes its comprehension and evolution more complex. Very few research works tackle this problem of dynamicity and heterogeneity while evolving DISS.

(c) Tools helping and supporting the user in the task of program-database co-evolution are almost unexisting. In particular, we observe that recommendation systems, providing users with automatic recommendations about what and where to change in the source code (at the line of code level), are cruelly missing.

Roadmap

This chapter has summarized the literature in the field of DISS evolution. We made interesting observations; in particular, we have observed that there is a severe lack of automatic support for DISS evolution.

The next chapter (Chapter 3) explores the use of the database schema evolution history as an additional information source to aid database schema reverse engineering.

Later, in Chapter 4, we present an analysis approach that aims to statically detect database accesses within the source and to recover their actual SQL values. In particular, this approach handles dynamic SQL as well as ORM technologies.

CHAPTER 3

UNDERSTANDING DATABASE SCHEMA EVOLUTION

This chapter[a] *explores the use of the database schema evolution history as an additional information source to aid database schema reverse engineering. We present a tool-supported method for analyzing the evolution history of legacy databases, and we report on a large scale case study of reverse engineering a complex information system.*

[a]This chapter extends two papers. The first one [Cleve et al., 2015] appeared in the journal *Science of Computer Programming* in 2015. The second paper [Meurice and Cleve, 2014] is a tool demo paper published in the proceedings of the IEEE CSMR/WCRE 2014 Software Evolution Week.

3.1 Introduction

Database reverse engineering (DRE) has traditionally been carried out by considering three main information sources: (1) the database schema, (2) the stored data, and (3) the application programs. Not all of these information sources are always available, or of sufficient quality to inform the DRE process. In recent years, the analysis of the evolution history of software programs has gained an increasing role in reverse engineering in general. Indeed, understanding the evolution history of a complex software system can aid and inform its current and future development. Software repositories - such as source code version-control systems, issue/bug-tracking systems, or communication archives -, where current and historical artefacts and interactions are registered, provide opportunities for historical analyses of system evolution (see Figure 3.1). The mining software repositories (MSR) field exploits the data available in software repositories. In recent years, many researchers have used MSR techniques as a way to support software understanding and evolution.

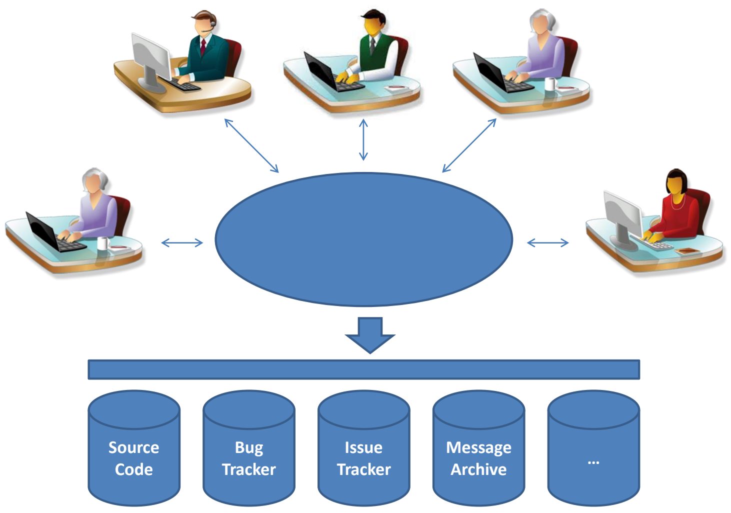


Figure 3.1: Software repositories.

Most research work in this area has concentrated on program code, design and architecture. But comparatively, little research has been carried out in the context of database reverse engineering. This is an unfortunate gap as databases are often at the heart of many of today's information systems. Understanding the database schema — which captures domain-specific concepts, data structures and integrity constraints — often constitutes a prerequisite to *evolving* such systems.

In this Chapter, we present a historical analysis approach that allows us to analyze the database schema evolution history by exploiting the versioning system.

3.2 Approach

Our approach consists of a *historical* analysis. That technique performs a comparison of all the successive versions of the database schema. Therefore, the created and deleted objects are emphasized. Our objective is to expose the evolution history of the database schema, through the study of the lifetime of all the database objects. Comprehending the past of the database schema can indeed significantly aid and inform current and future development activities (e.g., recovering missing integrity constraints, redocumenting the database schema, detecting bad design practices, etc.).

Furthermore, analyzing developers' activities involved in the development and maintenance processes, can considerably help at detecting the most qualified (group of) person(s) for achieving specific database-related maintenance activities.

In particular, our general approach consists in extracting and comparing the successive versions of the database schema from the versioning system, in order to produce the so-called **historical database schema**. The latter is a visual and browsable representation of the database schema evolution over time. It contains all database schema objects (i.e., tables, columns and constraints) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime, which in turn serve as a basis for the visualization of the schema and its further analysis. This historical schema can be queried in order to derive valuable information about the evolution of the database, potentially raising other interesting system-specific questions to investigate. The global process that we follow to build the historical database schema of a system consists of several steps. The approach overview is depicted in Figure 3.2.

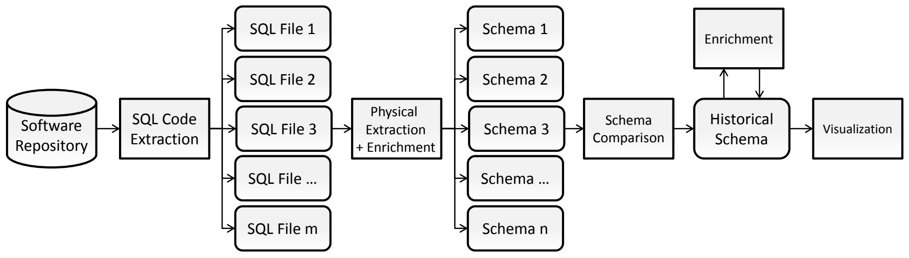


Figure 3.2: Approach overview.

3.2.1 SQL Code Extraction

The objective of the *SQL code extraction* phase is to extract, for each system version, all the SQL files constituting the complete DDL code required to fully create the database. This extraction phase is performed by exploiting the versioning system, which offers mechanisms for retrieving the different source code versions (including the database schema) saved/committed during the system development/maintenance. For each committed version, useful information are registered such as the accurate date of commit and the identity of the developer who committed the version; this person is called the *committer*. By retrieving and gathering all these information, questions like *who committed what and when?* can be answered.

3.2.2 Physical Extraction

When more than one DDL source file has been extracted for a single system version, the previous SQL code extraction phase results in several partial extracted schemas. The final physical schema must include the specifications of all these partial views, through a schema integration process. As output of this *physical extraction* phase, a unique physical schema is computed for each system version.

3.2.3 Physical Schema Enrichment

Referential integrity is an important quality attribute of data stored in relational information systems. It refers to the state where data stored in related tables obeys to the foreign key constraints defined between those tables. Most modern DBMS purposed for business applications provide features for declaring and automatically enforcing foreign key constraints. However, many legacy information systems do not use these features - or use them only to a limited degree (i.e., for more recently developed functionality), particularly if their original design predates availability of those mechanisms in the DBMS platform.

As previously discussed in Section 2.3.1, there exist several techniques to recover implicit properties/constraints of the database schema. Moreover, exploiting different information sources can help at inferring implicit (missing) foreign keys [Hainaut, 2002].

In our search for representing and comprehending the evolution of a database schema, inferring those implicit referential constraints and thus, enriching the extracted physical schema can bring important information. This problem is further addressed in Section 8.3.

3.2.4 Schema Comparison

The objective of the *schema comparison* phase is to compare the successive extracted physical schemas in order to incrementally derive the so-called *historical schema*. The historical schema is an integrated representation of the evolution of the database schema over time. It contains all database schema objects (i.e., tables, columns, keys and indexes) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime.

Let us now further illustrate the schema comparison step. The top side of Figure 3.3 gives an example of the evolution of a database schema, involving three successive schema versions. Schema S_1 is the oldest one and schema S_3 is the most recent one. One can see that between version 1 and version 2 column A_2 has been deleted, column B_2 has been created as well as table D and its columns. Moreover the entire table C has been dropped. In version 3, table B has disappeared, table D has been left unchanged, and table C has re-appeared. Indeed, it used to exist in version 1, it had been removed in version 2 and it is now back in version 3. We will refer to that phenomenon by saying that a schema object may have several lives.

The historical schema derived from the schema evolution example is depicted at the downside of Figure 3.3. The historical schema is a global representation of all previous versions of a database schema, since it contains all objects that have ever existed in the entire schema history. The historical schema is annotated with a list of couples $(date(S_i); committer(S_i))$ that provides, for each successive schema version S_i, the commit date $(date(S_i))$ and the identifier of the committer

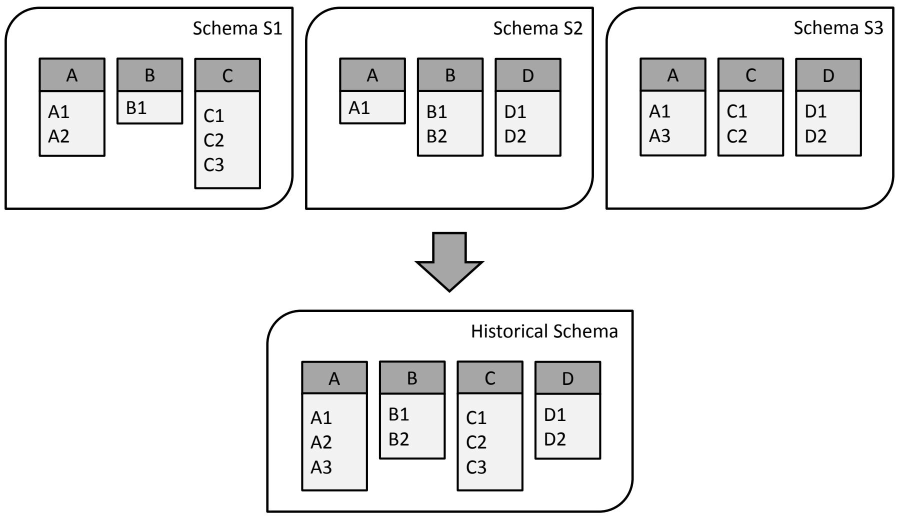


Figure 3.3: Schema evolution example (top) and corresponding historical schema (down).

($committer(S_i)$). For each object of the historical schema, we know the list of schema version dates where the object is present.

Algorithm 1 formalizes our procedure for deriving a historical database schema S_H from n successive schema versions. The historical schema derivation algorithm is based on a pairwise comparison of all those schema versions. The algorithm starts from an empty historical schema, and then iterates on all the schemas in chronological order, while comparing the current schema S_i with the current historical schema S_H. The comparison is made by iterating on each schema object (table, column, key or index) of both schemas. Several situations may occur for a given schema object:

(a) o belongs to S_i but does not belong to S_H. This means that o has been created in version i. We therefore add it to S_H and sets its list of presence to its version of creation, i.

(b) o belongs to S_i and belongs to S_H. In this case, we update its list of presence.

(c) o belongs to S_H but does not belong to S_i. In this case, the list of presence of o remains unchanged.

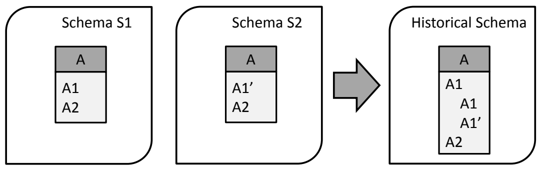


Figure 3.4: Example of a column modification between two schema versions and the corresponding historical.

As previously described, the historical schema is an integrated representation

```
1 Notations.
      • Let $S_i$ be a database schema version, defined as a set of schema objects
        (including a set of tables and their respective columns, keys and indexes).
      • Let $versions(o)$ be the list of presence of object $o$.

  Require: $S_1, S_2, \dots S_n$: $n$ successive schema versions.
  Ensure: $S_H$: the corresponding historical schema.
      // initializing $S_H$
   1: $S_H \leftarrow \emptyset$
      // iterating from the initial version to the latest version
   2: for all $i \in \{1 \dots n\}$ do
   3:    // objects $o$ appearing in more than one version
   4:    for all $o \in S_i \cap S_H$ do
   5:       $versions(o) \leftarrow versions(o) \cup i$
   6:    end for
   7:    // objects $o$ not present yet in the historical schema
   8:    for all $o \in S_i \setminus S_H$ do
   9:       $S_H \leftarrow S_H \cup o$
  10:       $versions(o) \leftarrow i$
  11:    end for
  12: end for
```

Algorithm 1: Deriving the historical schema from n successive schema versions.

of the database schema evolution. This historical representation can be queried in order to recover a precise knowledge of the evolution history of the database schema. Querying the historical schema can allow comprehending how the database schema has evolved over time. Hence, knowing the exact state of any schema object at anytime (i.e., at any system version) is primordial to understand its evolution. In particular, we also consider the column modification as an important historical knowledge. Column modification like a data type change of a particular column between two successive schema versions is detected and recorded in the resulting historical schema. Figure 3.4 depicts such a scenario where the data type of column A_1 changed between version S_1 and S_2. This modification is recorded and results in the creation of two *sub-columns* representing the state of A_1 before and after the modification. Those sub-columns have, in turn, their own list of presence.

3.2.5 Historical Schema Enrichment

Detecting a table/column renaming between two successive schema versions is an easy process when the SQL migration scripts are available. As illustration, the following SQL statement renames table A as B:

```
ALTER TABLE A
RENAME TO B;
```

Similarly, the following SQL statement renames column $A1$ of table A as $A2$:

```
ALTER TABLE A
RENAME COLUMN A1 TO A2;
```

However, in absence of such scripts, the task of detecting renaming becomes much more complex. Indeed, if table A is renamed as B, there is no direct way to detect it and the historical schema considers that table A has been dropped while table B has been created without keeping a link between both tables. In such a case, one sees that a finer-grained approach is required.

The *historical schema enrichment* phase consists of the automatic recovery of implicit table/column renamings in order to enrich the historical schema. One can formalize the problem of column renaming detection as follows:

•$S_1 \to \cdots \to S_n$, the n successive database schemas where S_1 is the initial schema and S_n is the latest (current) one.
•S_H, the historical schema.
•T, a table belonging to S_H.
•c_1 and c_2, two columns belonging to T

Two necessary conditions for a column renaming come out. If one supposes that c_1 was renamed as c_2 at schema version S_i, with $i \in \{2,\ldots,n\}$, then:

(a) $c_1 \in S_{i-1}, \notin S_i \land c_2 \notin S_{i-1}, \in S_i$
(b) $c_1 \simeq c_2$

The first condition ensures the presence of c_1 at the schema version predating its renaming, and its disappearance at the renaming version. Likewise, c_2 only appears once the renaming performed. The second condition means that c_1 and c_2 are syntactically different but remain semantically equal. However, those two conditions are not sufficient: one cannot detect a column renaming with certitude. The best we can do is affirming there is a column renaming according to a particular probability. This probability should represent the *similarity* value between c_1 and c_2. The greater this value, the more similar the columns. Therefore, we can define the function *column_similarity*:: $Column \times Column \to [0,1]$, taking as input two columns and returning the similarity value. This function can rely on several metrics, such as:

- The similarity between the column names. Several existing algorithms address this issue of *string* similarity. In particular, string distance algorithms measure the similarity between two strings (e.g., Levenstein, Monge-Elkan, Smith-Waterman, Jaro-Winkler distances) [Cohen et al., 2003].
- The similarity between the column types. The similarity between two columns can also be measured by considering their datatype, cardinality, default value, etc. Also, if the columns are part of identifiers, indexes, foreign keys, etc.

In summary, we could thus define the function *column_similarity* as follows:

- column_similarity(c_1,c_2) =
  α.stringDistance($c_1.name$, $c_2.name$) + β.type_similarity(c_1,c_2)
- $\alpha + \beta = 1$

such as α and β represent the weighting of each similarity criterion.

Finally, if the similarity value between c_1 and c_2 is equal to or greater than a certain minimal acceptance value, c_2 can be considered as the renamed version of c_1. Let us illustrate it with a concrete example. Let us suppose we search for the implicit column renamings performed at schema version k, with $k \in \{2, \ldots, n\}$. Let,

- $c_k = \{c_{k_0}, \ldots, c_{k_s}\}$, the set of columns of the table T created at version k
- $d_k = \{d_{k_0}, \ldots, d_{k_t}\}$, the set of columns of the table T dropped at version k
- $p \in [0, 1]$, the minimal threshold

We would like to detect the set of implicit column renamings performed at version k. We can firstly compute the matrix C such as,

$$C = \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0t} \\ c_{10} & c_{11} & \cdots & c_{1t} \\ \vdots & \vdots & \ddots & \vdots \\ c_{s0} & c_{s1} & \cdots & c_{st} \end{pmatrix}, \text{ such as}$$

$$(1)\quad c_{ij} = \delta_p(c_{k_i}, d_{k_j}) \times column\_similarity(c_{k_i}, d_{k_j})$$

$$(2)\quad \delta_p(a, b) = \begin{cases} 1, \text{ if } column\_similarity(a, b) \geq p, \\ 0, \text{ otherwise} \end{cases}$$

A first way to address the problem of column renaming detection is to retain the greatest similarity values respecting the minimal acceptance rate. However, in some cases, this naïve solution could miss some renamings. Let us suppose that the minimal acceptance value is fixed at $p = 0.8$ and the C matrix has the following form:

	x	**y**
a	0.9	0.85
b	0.8	0.2

The maximal value $(a, x) = 0.9$ is retained, at the detriment of the couples (a, y) and (b, x) which are inferior and thus skipped (since at most one couple per row and column can be retained). The only remaining possibility of renaming for b is the couple (b, y). Nevertheless, its similarity value 0.2 is less than the minimal acceptance rate 0.8 and is therefore rejected. As result, the couple (a, x) is the only detected renaming.

	x	**y**
a	0.9	0.85
b	0.8	0.2

However, through this example, we demonstrate that a more elaborated technique would be preferable. Indeed, the choice of the couples (a, x) and (b, y) instead of, for instance, the couples (a, y) and (b, x) is questionable.

Accordingly, we experimented a second technique which considers the present problem of renaming detection as an optimization problem subject to a set of constraints. Let X, the matrix of unknown factors such as

$$X = \begin{pmatrix} x_{00} & x_{01} & \cdots & x_{0,n} \\ x_{10} & x_{11} & \cdots & x_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m0} & x_{m1} & \cdots & x_{m,n} \end{pmatrix}$$

Our technique consists in achieving the best outcome in a mathematical model, namely the best composition of couples. This optimization problem can formalized as follows:

$$\boldsymbol{max \sum_{i=0}^{n} \sum_{j=0}^{m} c_{ij} x_{ji}}$$

such as

$$\begin{cases} (1) \ \ \forall i \in \{0,\ldots,n\}, \sum_{j=0}^{m} x_{ji} = 1 \\ (2) \ \ \forall j \in \{0,\ldots,m\}, \sum_{i=0}^{n} x_{ji} = 1 \\ (3) \ \ x_{ji} \in \{0,1\} \end{cases}$$

The first (1) and second (2) conditions guarantee that only one unknown factor per row and per column will be equal to 1. Column d_{k_j} is considered as renamed as column c_{k_i} if x_{j_i} is equal to 1; otherwise, the renaming is rejected.

This problem can be solved with linear programming (LP or also called linear optimization). LP is an optimisation problem in which the objective function is linear in the unknowns and the constraints consist of linear equalities and linear inequalities [Luenberger and Ye, 2015]. LP aims to achieve the best outcome for the optimization of a linear objective function, subject to linear (in)equality constraints. Linear programs are problems that can be expressed in the following standard form:

$$max\ cx,$$
$$Ax = b, \text{ with } x \geq 0$$

Our optimization problem can be easily adapted to LP with a few matrix transformations.

3.2.6 Visualization

The main aim of the historical schema is to propose a historical representation of database schema evolution. This historical representation can be exploited and queried in order to derive valuable information about the evolution of the database which could facilitate future initiatives and developments. Some interesting system-specific questions can be tackled by use of a visualization support. In particular, this visualization support, exploiting the historical schema as main information source, shall cope with some visual difficulties, like permitting an intuitive and efficient visualization of large database schemas and system histories.

3.3 Tool Support

We implemented a tool support allowing us to compute and visualize the historical schema in order to understand the database schema evolution history.

3.3.1 SQL Code Extraction

To achieve this task, we implemented a versioning system explorer which extracts information about each system version that have ever been registered/committed in the versioning system. Our explorer is currently designed to deal with SVN and GIT as versioning system. For each system version, we automatically extract (1) its identifier (generally a numeric/string value), (2) the date when the version was committed (`MM/DD/YYYY HH:MM:SS` format), (3) the identifier of the developer who committed the version (an example of encountered developer ids is her/his name or email address) and (4) the SQL files (.sql, .ddl, ...) present in the version.

However, in order to be consistent and complete in this process, the assistance of the user may be required for selecting the relevant SQL files to conserve/reject. Indeed, the user is free to decide which files are (not) part of the database schema. This pre-filtering phase is often necessary to avoid unexpected results which might alter/pollute the actual database schema (e.g., migration SQL scripts, DDL code for other platforms/DBMS, etc.).

Once the SQL file selection is confirmed, our tool merges the content of the retained files, for each version. As output, we obtain a concise SQL/DDL file for each system version.

3.3.2 Physical Extraction

The objective of the *physical schema extraction* phase is to extract the physical schema corresponding to each SQL file obtained so far.

A first attempt of automated physical extraction was tried by use of the DB-Main CASE tool [1]. DB-Main is a tool dedicated to the database engineering domain. DB-Main proposes several tools and in particular, it offers a pretended technology-independent SQL parser which takes as input a DDL script and produces the corre-

[1]DB-Main, http://www.db-main.be.

sponding physical schema, expressed in a proprietary format. However, based on our experience, we encountered severe problems due to some parsing weaknesses.

- The SQL code resulting from the SQL code extraction phase does not always fit the SQL parser of DB-Main. As illustration, we learned that the grammar used by DB-Main only covers a subset of the MySQL grammar and other DBMS's grammar. Unfortunately the parser does not ignore the word when it does not know it. It skips the entire line or even the entire table definition.
- The physical schema resulting from the DB-Main parsing phase is expressed in a proprietary format and is either directly visualizable in DB-Main itself, or manipulable via the API furnished JIDBM. However, we observed some limitations of DB-Main when dealing with a big number of large database schemas, in particular the response-time when loading them in memory.

Whereas we understand the complexity of implementing a complete DBMS-independent SQL parser, we opted for a *semi-automatic* solution which is, in our opinion, a balanced solution. We developed, in collaboration with the DB-Main developers, a program (independent of DB-Main) taking as input the DDL code and returning the corresponding physical schema. To achieve its task, the program initially creates the target database by connecting the DBMS and executing the DDL code. Once the database is created, the program accesses the metadata of the latter (e.g., catalogs, schemas, table structures) and finally extracts its schema, tables, columns and constraints (i.e., primary/unique/secondary/foreign keys, indexes, etc.). The extracted physical schema is expressed in an XML form.

While this program is fully automatic, it nevertheless requires an installed and running DBMS which is compatible with the DDL code to execute. This process currently supports several DBMS, i.e., MySQL, Oracle, DB2 and PostgreSQL.

We then run this process on each system version to obtain the successive physical schemas.

3.3.3 Physical Schema Enrichment

We designed an automatic tool-supported approach for detecting implicit referential constraints (i.e., foreign keys) by considering new information sources. This approach, presented later in Section 8.3, permits us to enrich the extracted physical schemas.

3.3.4 Schema Comparison

The challenge we face by implementing our historical schema derivation tool is to be sufficiently scalable to analyze long histories and large database schemas in satisfactory time. We therefore implemented a multi-threaded version of Algorithm 1. This multi-threaded version separately handles each successive schema version (see Figure 3.5). For each schema version, an independent thread is created per encountered table. The thread is responsible for deriving the historical information about that table and its content (columns, keys and indexes). All parallel threads share a common resource, namely the historical schema, that they can all update when they discover information about their respective tables. The main program thread iterates over all successive schema versions and tables. This multi-threaded implementation can constitute a significant performance improvement of the efficiency of our historical schema derivation tool, in presence of multi-core processors.

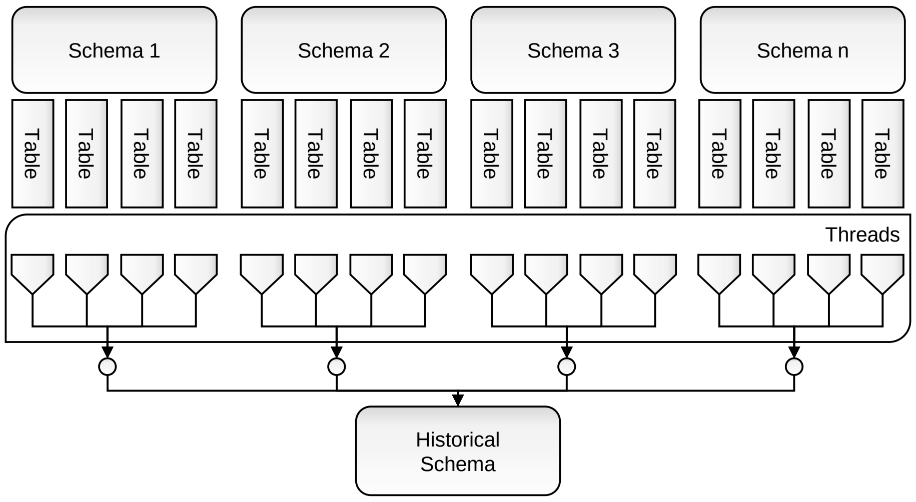


Figure 3.5: Multi-threaded implementation of our historical schema derivation tool.

3.3.5 Historical Database Schema Enrichment

We designed a semi-automatic tool approach supporting the identification of implicit (column/table) renamings. Our tool computes the *most likely* implicit renamings according to different comparison criteria (e.g., name similarity, the column type similarity, etc.). To achieve this objective, we implemented our tool based on the linear programming method which aims at achieving the best outcome in the mathematical model discussed in Section 3.2.5. The weight of each metric used to define the *similarity* function can be configured according to the user's will, as long as the weighting sum is equal to 1.

However, we are aware that further experiments are required to prove the suitability and efficiency of this technique and to validate this choice (instead of other techniques).

3.3.6 Visualization

We implemented **DAHLIA** (DAtabase ScHema EvoLutIon Analysis), an interactive visualization tool of database schema evolution. DAHLIA provides the user with a visual and browsable representation of the database schema evolution history. It takes the historical schema as input and allows one, among others, to (1) compare two arbitrary schema versions, (2) extract the database schema at a given date, (3) extract the complete history of a particular schema object (column/table), (4) extract various statistics about the evolution of the database schema, (5) analyze the involvement of each developer in that evolution.

During the development of DAHLIA, we explored two different types of visualization, namely a 2D and 3D visualizations.

2D mode

This mode proposes an interactive panel allowing one to manipulate the physical objects of the historical schema and query their respective history. In this mode, the historical schema is represented and displayed according to the entity-relationship (ER) model. Figure 3.6 depicts an example of schema visualized in **DAHLIA** and expressed as an ER schema. This schema includes several physical objects such as tables, columns, identifiers (ID), foreign keys (FK) and indexes (INDEX).

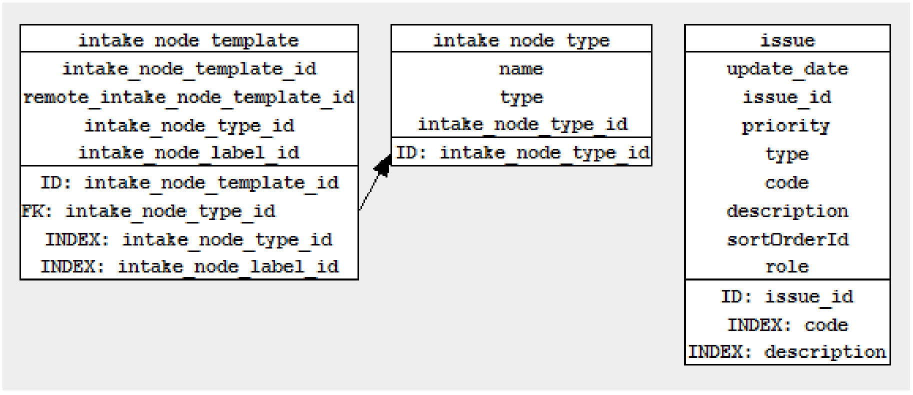


Figure 3.6: An example of schema as visualized in DAHLIA 2D.

Several functionalities are offered to users for querying the historical schema.

Colourizing the historical schema. DAHLIA offers diverse features for extracting historical knowledge about schema evolution. The first one gives a first insight about the longevity of the physical objects. Our tool assigns a colour to each object depending on its age and liveness. All objects depicted in green constitute the objects which are still present in the latest schema version, while all red objects have been dropped. The colour shades corresponds to the age of the objects. A dark red object is an object that has been dropped a long time ago, whereas a light red (orange) object is an object that has recently been removed from the schema. The darker the green, the older the object is, and vice versa. Figure 3.7 depicts an example of historical schema which is colourized according to this color pattern.

Filtering. DAHLIA proposes the user a filtering functionality allowing displaying the database schema version corresponding to an input date/commit. In this way, the user can accurately know the state of the database schema at any moment of the development/maintenance phase.

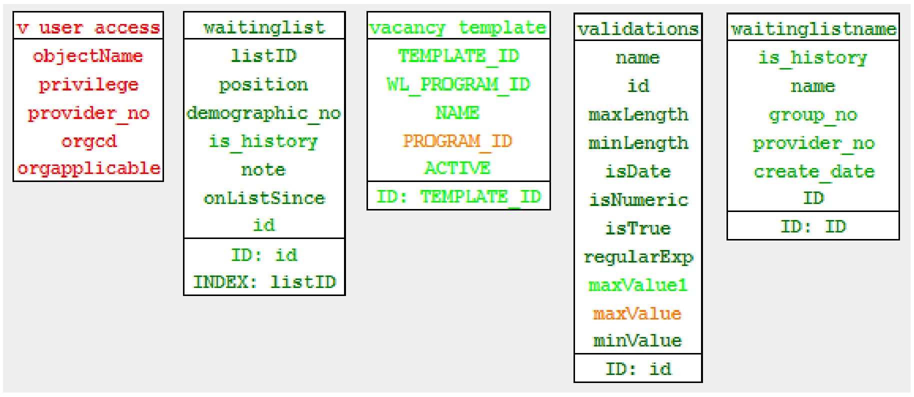
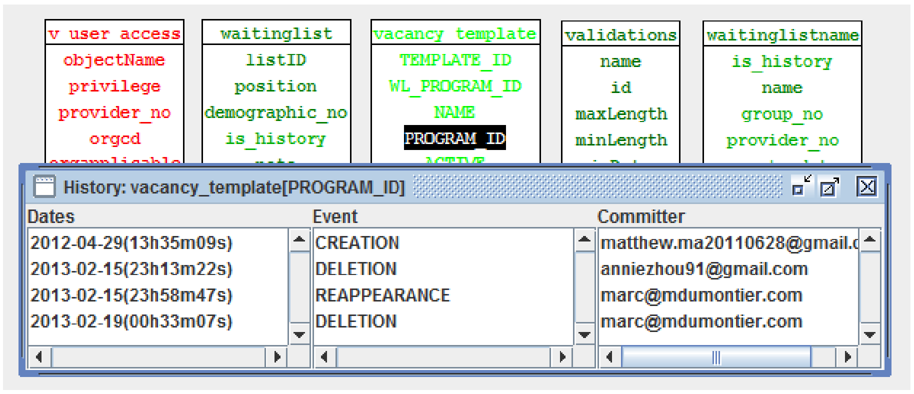


Figure 3.7: An example of colourized historical schema as visualized in DAHLIA 2D.

Displaying the complete history of a particular schema object. In addition, the history of each single schema object is also available via our tool. This historical information includes the date(s) of all creation, update and deletion operations, and for each of them, the id of the corresponding developer.



History: vacancy_template[PROGRAM_ID]

Dates	Event	Committer
2012-04-29(13h35m09s)	CREATION	matthew.ma20110628@gmail.c
2013-02-15(23h13m22s)	DELETION	anniezhou91@gmail.com
2013-02-15(23h58m47s)	REAPPEARANCE	marc@mdumontier.com
2013-02-19(00h33m07s)	DELETION	marc@mdumontier.com

Figure 3.8: The complete history of column *PROGRAM_ID* displayed in DAHLIA.

DAHLIA allows an in-depth analysis of the evolution of a given schema object. Figure 3.8 shows the complete history of column *PROGRAM_ID*. The column has been created on 29 April 2012 by the developer identified by *matthew.ma20110628@gmail.com*. Then, the column has been removed on 15 February 2013 by *anniezhou91@gmail.com* and has immediately been recreated on the same day by *marc@mdumontier.com*. The column has been finally dropped on 19 February 2013 by the same developer. This evolution history gives additional information about how the object has evolved over time.

Analyzing developers' involvement. DAHLIA allows analyzing developers' activities on the database schema. By *activity*, we mean any performed database schema change such as:

- Adding a table
- Dropping a table
- Adding a column
- Dropping a column
- Adding an identifier
- Dropping an identifier
- Adding a foreign key
- Dropping a foreign key
- Adding an index
- Dropping an index
- Changing a column datatype
- Renaming a table
- Renaming a column

Figure 3.9 shows the activities of each developer, as visualized in DAHLIA. The bar chart depicts the activities of each developer in terms of number of schema changes. This chart aims to show up the *database specialists*. The pie chart shows the proportions of schema change types occurred in the system life.

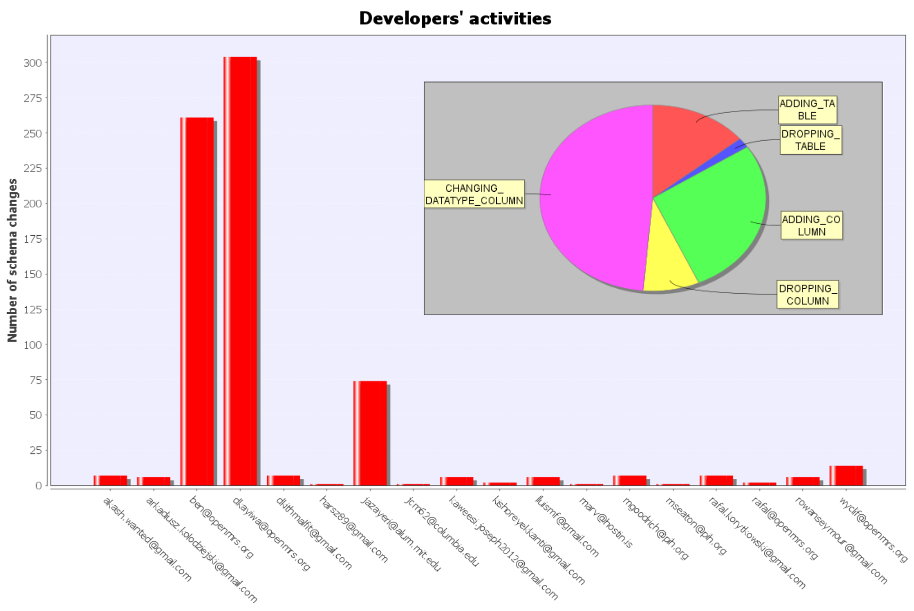


Figure 3.9: The charts of developers' activities as displayed in DAHLIA.

Moreover, the schema changes performed by a given developer can be precisely analyzed and displayed. Other features permit to reduce the search scope and thereby, filtering developers' activities on a particular table. Those features aim at answering some primordial evolution questions such as *"who are the specialists of that table/part of the database?"*, *"who is the most appropriate (e.g., most experienced or most qualified) person for achieving a particular database-related activity?"*.

Extracting evolution statistics. We also provide users with a historical schema querying tool, allowing the extraction of interesting statistics regarding the evolution of the schema of interest (the evolution of number of columns/tables, the evolution of the average number of columns per table, the number of columns/tables created/deleted per version, ...).

Although the 2D visualization mode permits users to obtain precise details about the historical database schema structures, it nevertheless has some drawbacks. In particular, that 2D visualization is inconvenient for analyzing very large database schemas and long histories. Indeed, visualizing database schemas containing hundreds of tables and thousands of columns may prove complicated and may make the analysis of such schemas laborious. In addition, this visualization mode does not facilitate understanding how the schema has evolved over time. Figure 3.10 illustrates this issue when visualizing the historical schema containing several hundreds of tables and columns. This schema was obtained by applying our approach to a real-life subject system.

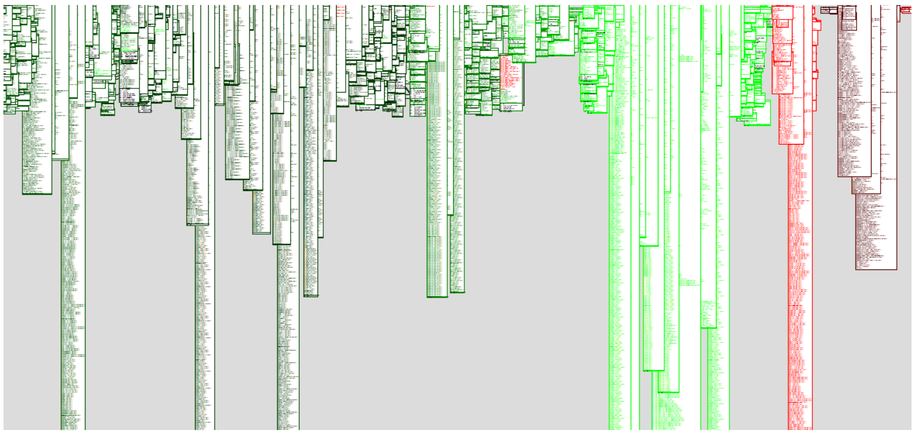


Figure 3.10: 2D Historical schema as visualized in DAHLIA.

Those observations, resulting of the use of the 2D mode during the in-depth study of actual systems with large database schemas, outlined those weaknesses and encouraged the implementation of a new visualization mode.

3D mode

In order to address those weaknesses, we implemented a 3D visualization mode which offers new functionalities.

The city metaphor. The 3D mode makes use of the well-known city metaphor of *CodeCity* introduced by [Wettel and Lanza, 2008a,b]. *CodeCity* is a language-independent interactive 3D visualization tool for the analysis of large software systems. Based on the city metaphor, it depicts classes as buildings and packages as districts of a *software city*. The authors demonstrated the efficiency and effectiveness of city-metaphoric visualization in the domain of program comprehension and software evolution analysis. They conducted an extensive controlled experiment over a period of six months in several countries, with academics and industry practitioners [Wettel et al., 2011].

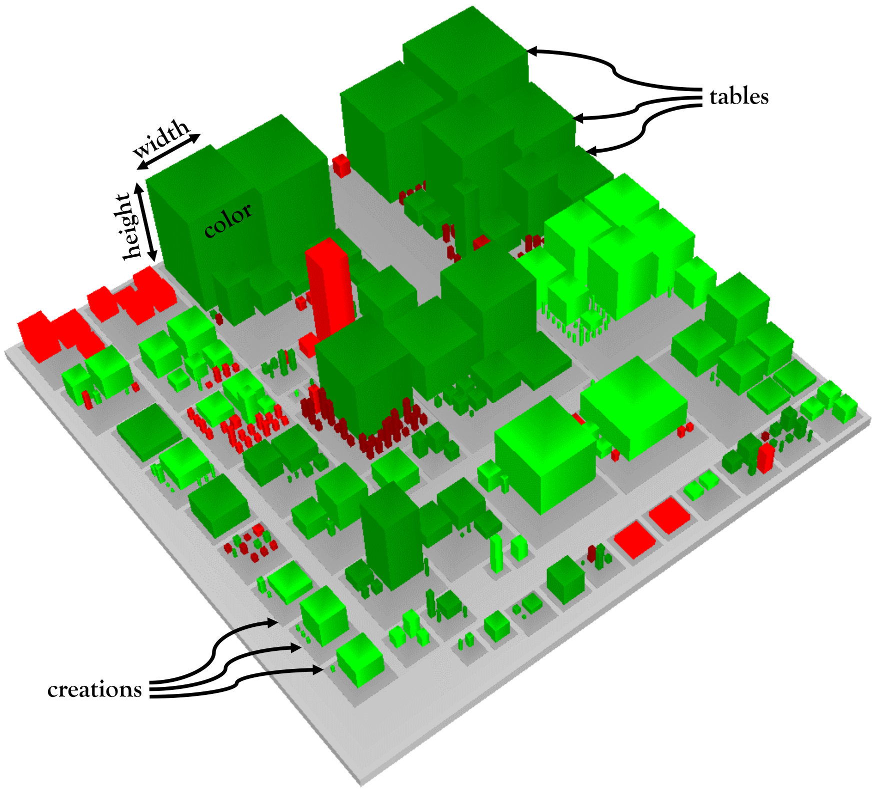


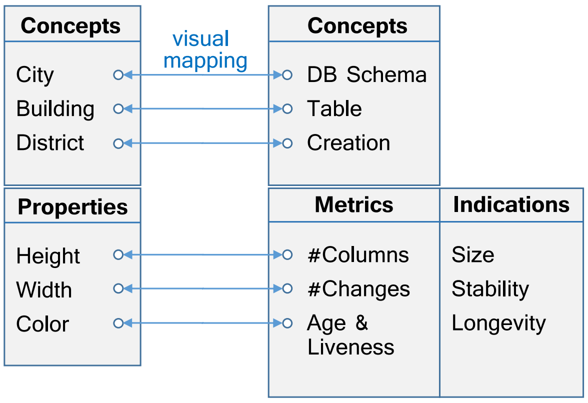
Figure 3.11: Historical schema visualized in DAHLIA as a 3D city.

In DAHLIA, a city represents the database schema and the buildings represent the tables belonging to the schema. Another visualized concept, pertaining to the table lifetime, is its creation date/version. All the tables with a similar creation version are gathered in a same district. By use of this metaphor, several visual metrics (summarized in Table 3.1) can be exploited:

(a) Building height: the building height is mapped on the number of columns. This visual metric aims at giving indications about the table size and, thus, allows directly detecting the massive and important tables.
(b) Building width: the building width (i.e., the building base size) represents the number of schema changes happened during the table life. This visual metrics aims at giving indications about the stability of each table.

(c) Building color: a color is affected to each building according to the *colourization pattern* applied to the historical schema. This visual metric aims at giving indications about the longevity of each table.

The mapping between the database schema metrics and the visual metrics can be (re)configured by the user. The user is naturally free to affect any *width* metric to the building *height* and vice-versa. Table 3.1 only represents an "advised" metric affectation. By applying those visual metrics to our historical schema, we obtain the 3D city representation as visualized in Figure 3.11.



Concepts	visual mapping	Concepts	
City	↔	DB Schema	
Building	↔	Table	
District	↔	Creation	
Properties		**Metrics**	**Indications**
Height	↔	#Columns	Size
Width	↔	#Changes	Stability
Color	↔	Age & Liveness	Longevity

Table 3.1: The visual mappings used in the city metaphor.

Visualizing database schema evolution. Likewise the 2D mode, DAHLIA 3D proposes the user a filtering functionality to display the database schema version corresponding to an input date/version. Furthermore, a *schema-diff* feature is also implemented and gives the opportunity to compare two different schema versions, which was a tedious task with the 2D mode. It helps the user to intuitively visualize how the schema has evolved over time. Figure 3.12 illustrates an example of schema-diff between two given schema versions. We implemented a *side-by-side* visualization where the left city represents the oldest selected schema version and the right city represents the most recent selected schema version. This dual visualization uses its own colour pattern:

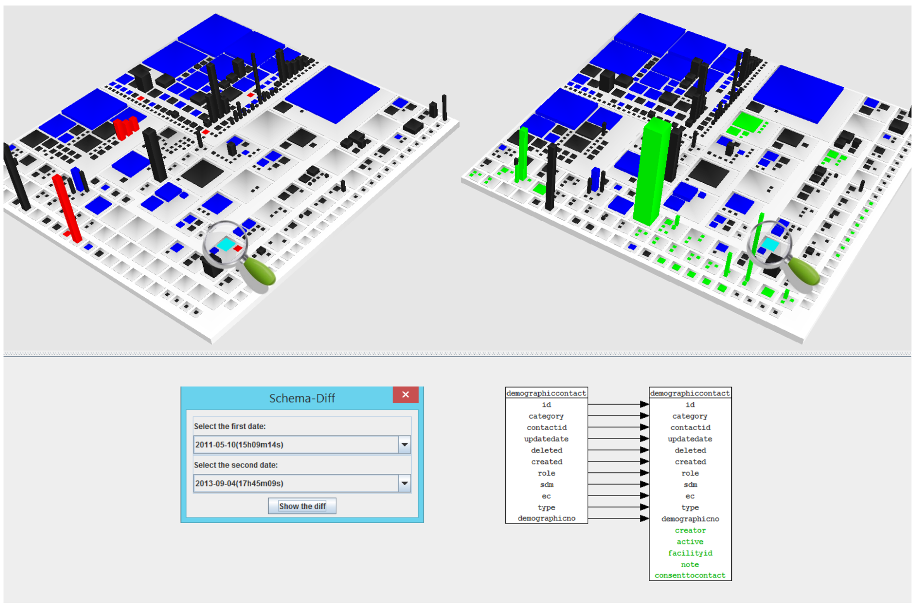
(a) A **black** building represents a table which remains unchanged between both versions, namely a table present in both versions. Therefore, a black building has an *identical* twin in the opposite city.

(b) A **red** building represents a table present in the left city (oldest version) but absent/removed from the right city (most recent version). It thus represents a table deleted between both versions.

(c) A **green** building represents a table present in the right city but absent from the left one. It thus represents a table added between both versions.

(d) A **blue** building represents a table present in both versions but which was modified (renamed, adding/deletion/renaming of columns/keys/indexes, ...).

A blue building has a *fraternal* twin in the opposite city.


Schema-Diff

Select the first date: 2011-05-10(15h09m14s)

Select the second date: 2013-09-04(17h45m09s)

Figure 3.12: Example of schema-diff between two particular schema versions.

In addition, a finer analysis of the schema differences at the table scale is allowed; clicking on a particular building of a city will (1) highlight, if present, the twin table in the opposite city, and (2) display the *table-diff* on an information panel. In Figure 3.12, the user has clicked on a particular blue table of the left city; the twin table has been highlighted in the right city. Also, a 2D representation of the *table-diff* is provided, using the same colour pattern - in this case, one observes the addings of several columns between both versions.

Another interesting implemented feature is the possibility to have an overview of how the database schema has evolved over time. For that, we conceived a "time traveller" (see Figure 3.13) which allows us to successively display all the schema versions - and thus, cities - of the studied system. With this *time traveller*, users can control and visualize the *movie* of the evolution of the database schema and observe phenomenons like the *destruction* of entire districts, the *appearance* of massive buildings, the *renovation* (modification) of buildings, etc.

Visualizing developers' activities. This 3D representation of a database schema provides us with new possibilities to visualize developers' activities. We implemented a new functionality which aims to project the specialization level of a given developer on the city. Figure 3.14 shows the specialization level of a given developer. This specialization level is expressed according to a *shading scale*. This scale aims to colourize the database tables according to the modifications performed by a given developer. Light pink buildings represent tables barely impacted by this devel-

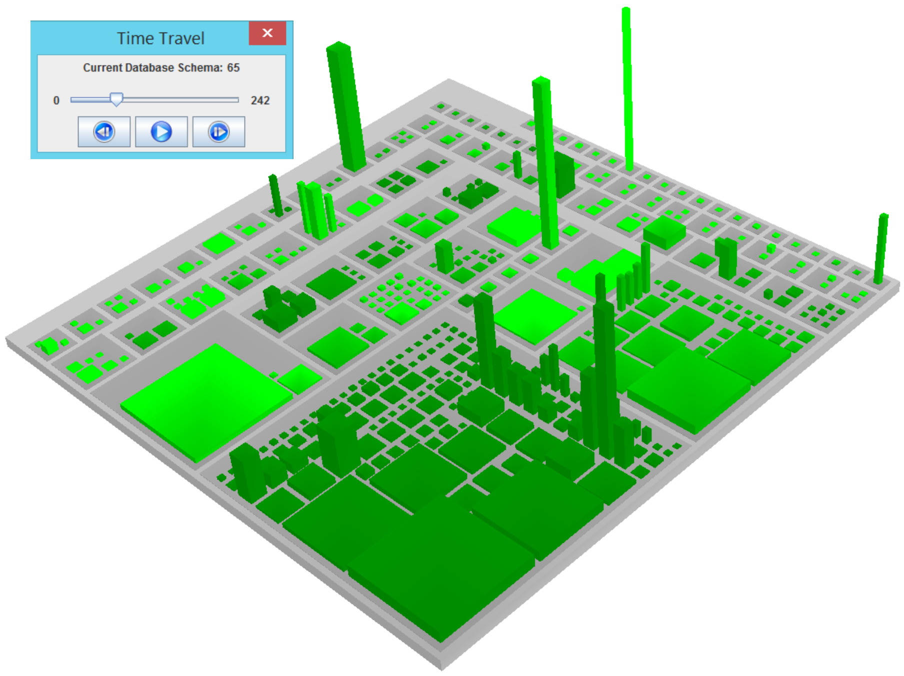


Figure 3.13: Time travel - Successive display of the different schema versions.

oper (i.e., fewer modifications performed on those tables), whereas red buildings represent tables which were significantly impacted by this developer.

Such a visualization facilitates detecting the expertise domain of each developer.

3.4 Application To Real-Life Systems

We selected six popular open-source database applications from different domains, and we used our historical schema approach and DAHLIA to visualize the evolution of their database schema.

(a) OSCAR: OSCAR[2] is full-featured Electronic Medical Record (EMR) software system for primary care clinics. It is widely used in hundreds of clinics across Canada. The OSCAR system is further discussed in Section 3.5.

(b) MediaWiki: MediaWiki[3] is a free and open source wiki software, used to power wiki websites such as Wikipedia, Wiktionary and Commons, developed by the Wikimedia Foundation and others.

(c) TikiWiki: TikiWiki[4] is a free and open source wiki-based, content management system and Online office suite.

[2]http://oscar-emr.com/
[3]https://www.mediawiki.org/wiki/MediaWiki
[4]https://tiki.org/tiki-index.php

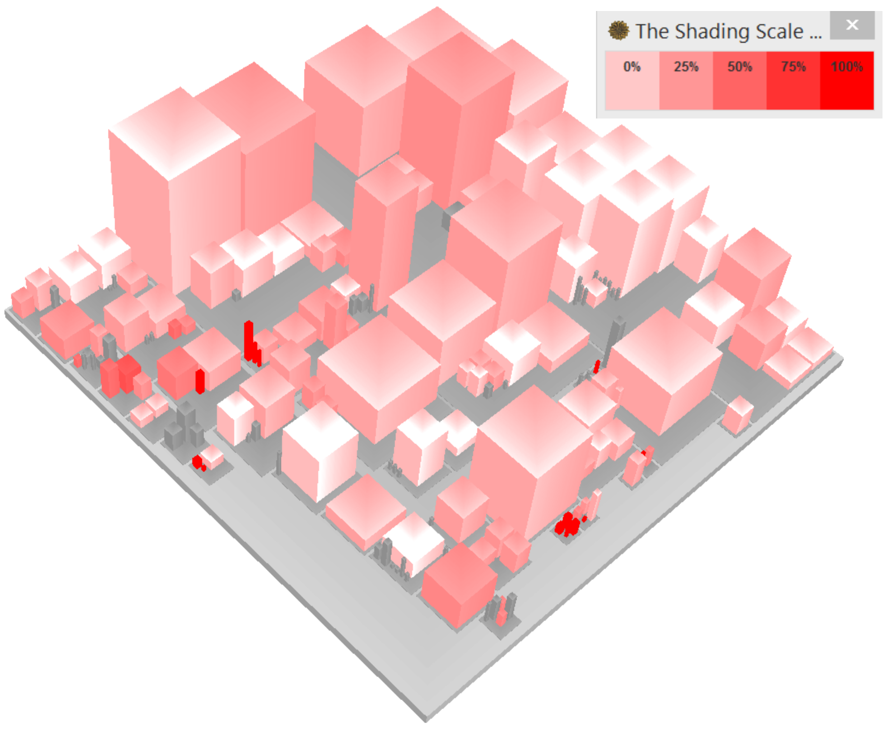


Figure 3.14: Visualization of the activities of a given developer projected on the city.

(d) PrestaShop: PrestaShop[5] is a free, open source e-commerce solution.
(e) OpenMRS: OpenMRS[6] is a collaborative open-source project to develop software to support the delivery of health care in developing countries (mainly in Africa).
(f) Broadleaf Commerce: Broadleaf Commerce[7] is an open-source, e-commerce framework.

Table 3.2 describes the six subject systems, the studied period and their evolution. The history of OSCAR, MediaWiki, OpenMRS and Broadleaf has been extracted from their respective GitHub repository, while PrestaShop and TikiWiki use SVN as version control system. As previously discussed, our tool approach allows the automatic extraction of the schema history from those two popular version control systems, i.e., Git and SVN. Table 3.3 shows, for each project, the distribution of the atomic database change types. The most frequent operation is the modification of the column datatype. Furthermore, one notices the general evolution trend is adding new structures, especially columns, tables and indexes. The developers rarely

[5] https://www.prestashop.com/
[6] http://www.openmrs.org/
[7] http://www.broadleafcommerce.com/

remove existing data structures. The figures of Table 3.3 have been automatically produced by DAHLIA.

Project	Studied Period	#Tables	#Columns	#Versions
OSCAR	07/2003 → 06/2013	88 → 445	2443 → 13364	670
MediaWiki	05/2003 → 08/2013	17 → 50	100 → 337	359
TikiWiki	12/2006 → 07/2013	206 → 248	1525 → 1974	623
PrestaShop	12/2008 → 09/2012	113 → 157	564 → 890	229
OpenMRS	12/2007 → 04/2015	72 → 88	564 → 950	164
Broadleaf	02/2019 → 03/2015	24 → 178	145 → 987	118

Table 3.2: The 6 studied database applications and their evolution.

Change type (%)	OSCAR	MediaWiki	TikiWiki	PrestaShop	OpenMRS	Broadleaf
Adding a table	9.7	9.6	19.4	19.6	3.3	9.5
Dropping a table	1.5	3.9	3.4	1.7	1.4	5.5
Adding a column	28.7	15.7	14.7	14.9	23.7	17.2
Dropping a column	3.5	5.4	2.5	2.6	6.8	13
Adding an ID	0.8	2.5	1.6	2.7	0.6	2.4
Dropping an ID	0.3	0.9	1.4	0.7	0.5	2.5
Adding a FK	0.05	0	0	0	5.7	5.4
Dropping a FK	0.2	0	0	0	2.5	5.4
Adding an index	2.3	12.5	5.4	14.7	8.3	10.8
Dropping an index	0.4	4.3	2.3	2.3	3	8.9
Changing a col. type	41.6	44.1	48.8	39.6	41.8	18
Renaming a table	0.2	0.11	0.1	0.1	0.2	0.8
Renaming a column	10.6	0.9	2.5	1	1.8	0.6

Table 3.3: Distribution of schema changes across the six subject systems.

Figures 3.15, 3.16, 3.17, 3.18, 3.19 and 3.20 depict the 3D historical schemas of, respectively, OSCAR, MediaWiki, TikiWiki, PrestaShop, OpenMRS and Broadleaf. Each city has its own characteristics and architecture.

- OSCAR: the OSCAR city combines very high buildings with flat fields, what depicts a quite strong inequality within the schema, in terms of size (some tables have more than thousand columns). Moreover, the large majority of the oldest tables are still existing in the latest studied schema version. However, the schema has undergone significant changes during its lifetime, with the successive adding of table districts over time which, in turn, have evolved. As example, one can observe that some of the highest tables were only recently created.
- MediaWiki: the MediaWiki city presents a smaller and more balanced schema in terms of size, with a maximum of 26 columns per table. Moreover, the latest schema version is, in majority, composed of the oldest tables. Indeed, the schema has only undergone small changes over time, with small additions of tables between versions.
- TikiWiki: the TikiWiki city presents a balanced mix of small and high tables (a maximum of 62 columns per table). The highest tables are also the oldest

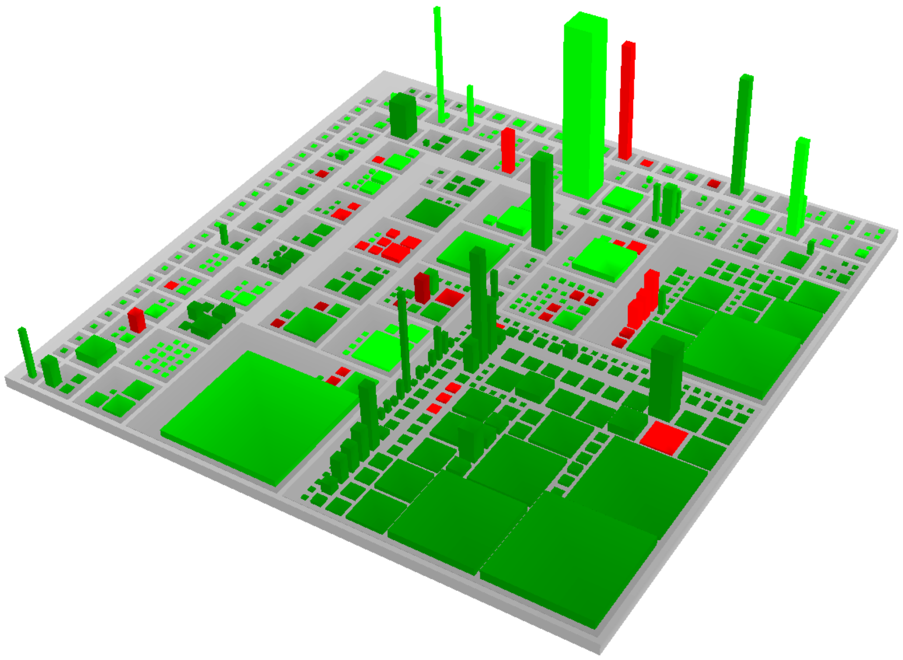


Figure 3.15: 3D Historical schema of OSCAR.

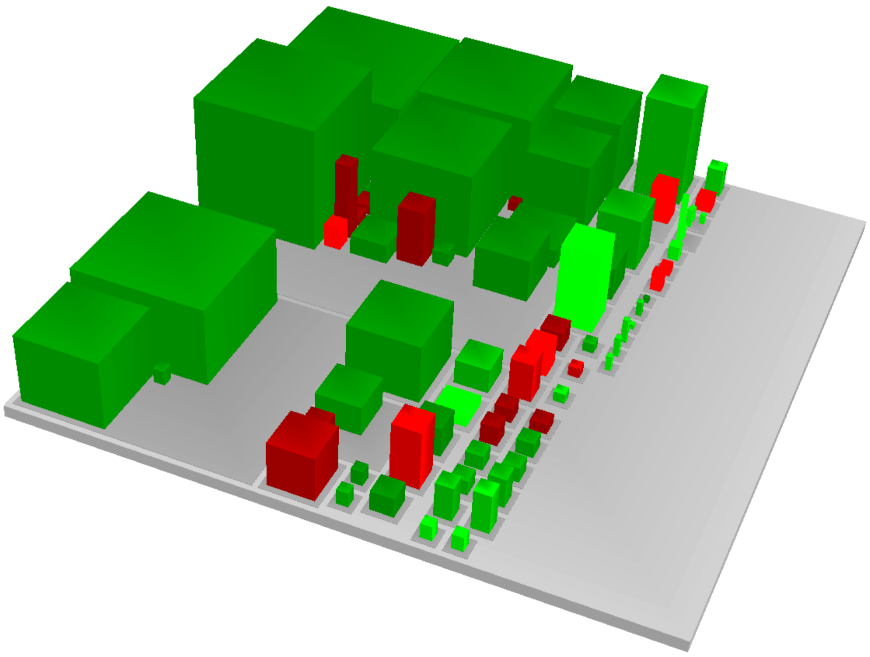


Figure 3.16: 3D Historical schema of MediaWiki.

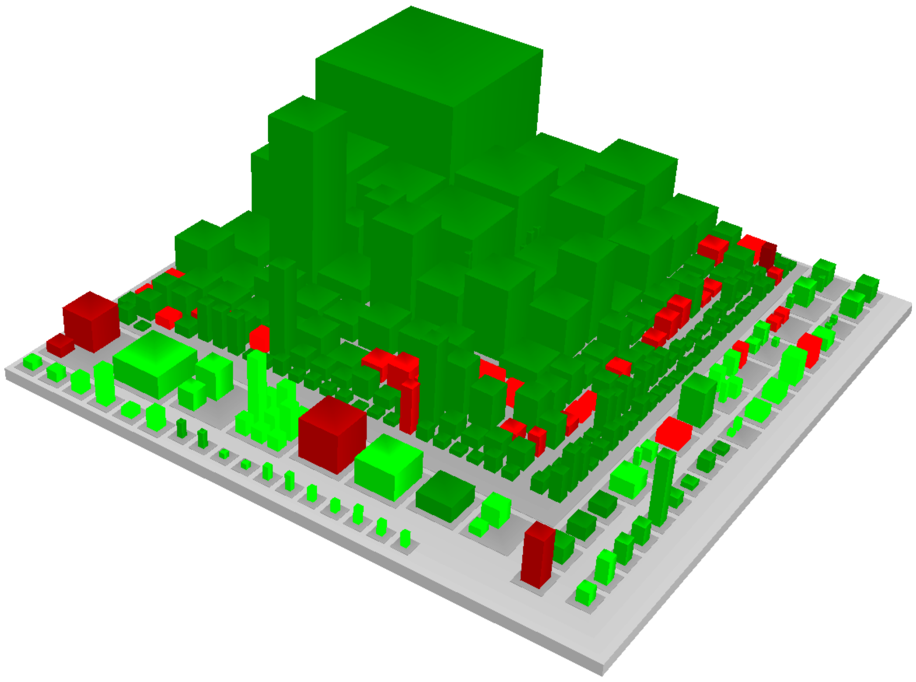


Figure 3.17: 3D Historical schema of TikiWiki.

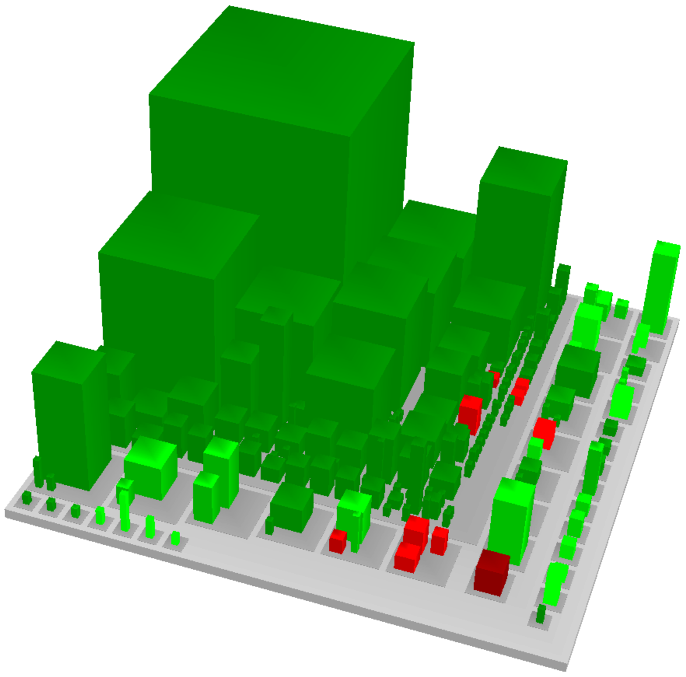


Figure 3.18: 3D Historical schema of PrestaShop.

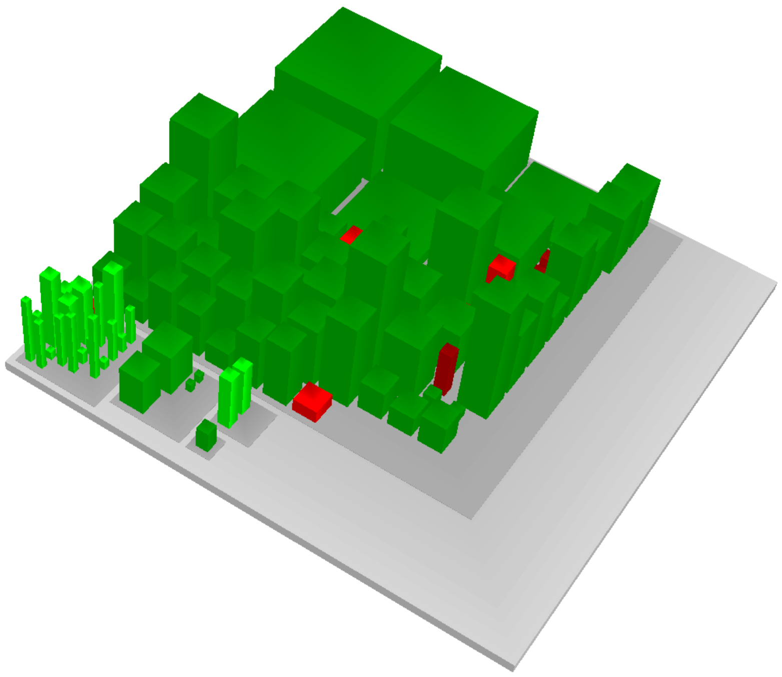


Figure 3.19: 3D Historical schema of OpenMRS.

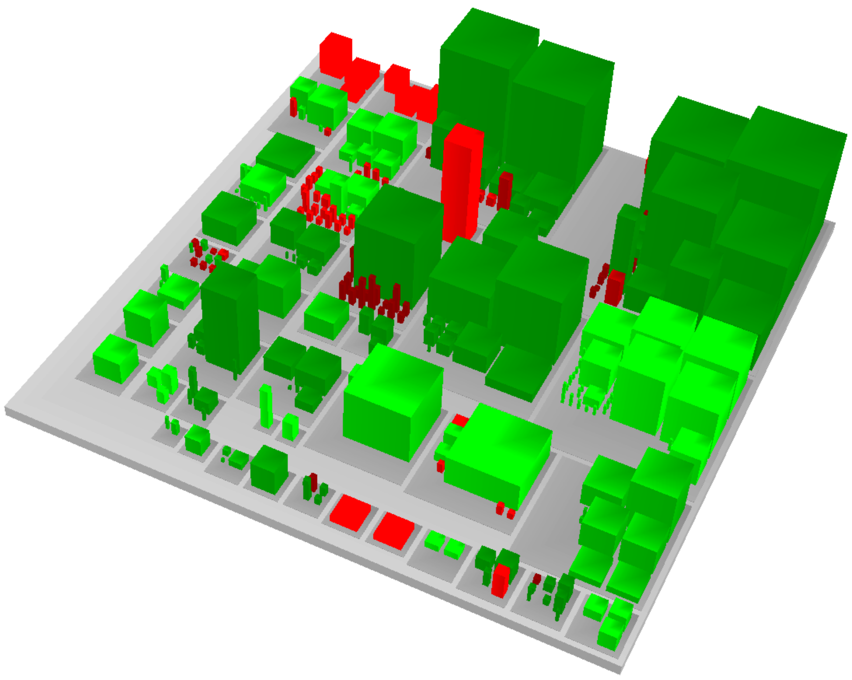


Figure 3.20: 3D Historical schema of Broadleaf.

ones. Although some of the oldest tables were removed, most of them are still existing in the latest schema version.

- PrestaShop: the Presthop city presents a quite similar architecture to TikiWiki, with a mix of small and high tables (a maximum of 50 columns per table). Like TikiWiki, the oldest tables are the highest ones and are still existing in the latest schema version. Only fewer tables were removed over time. The schema has only undergone the addition of few small tables.
- OpenMRS: the OpenMRS city depicts a quite homogeneous schema where the column number of every table seems quite similar. The creation of new tables is very rare, with only 5 districts of tables. Furthermore, very few tables were removed over time. In general, one observes a certain schema stability, with little changes.
- Broadleaf: the Broadleaf city presents a balanced schema, in terms of size (a maximum of 30 columns per table). However, unlike the other 5 cities, one can observe the removal of a certain number of tables, especially among the oldest tables.

Logically, we observe a common property to every city; the oldest tables mainly represent the less stable tables, i.e., the tables with the higher number of changes.

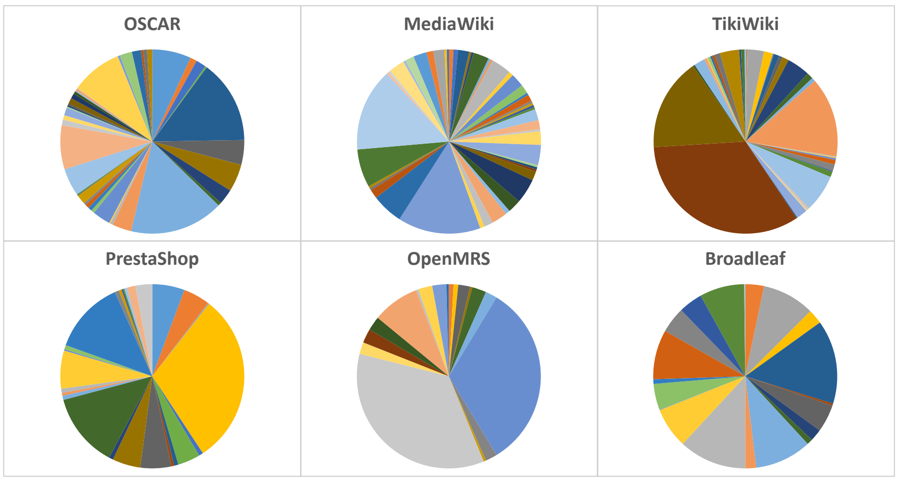


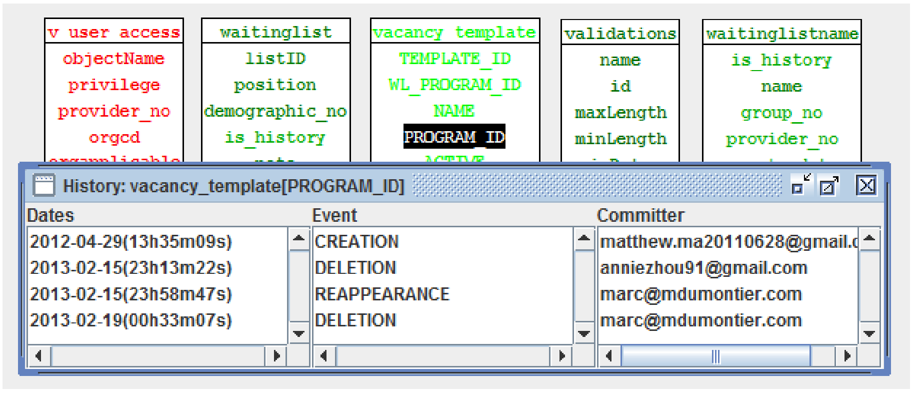
Figure 3.21: Information about the developers involved in the evolution of the database schema - a slice corresponds to the number of tables impacted by a developer (one slice per developer).

Figure 3.21 shows the involving of the developers of each studied project. Each slice corresponds to a developer and its size represents the number of distinct tables impacted by the developer (by creating, updating or deleting the table). Those figures have been automatically generated by DAHLIA. One can see the distribution of the OSCAR, MediaWiki and Broaleaf developers is quite homogeneous (fewer specialists), whereas a big part of the database changes of TikiWiki, PrestaShop and OpenMRS is performed by a single person (two people in OpenMRS).

Distributing responsibility among developers: which strategy?

During its lifetime, a system is subject to continuous modifications and its database schema undergoes structural changes. Those schema changes, required to keep systems adapted to ever-changing business and technological needs, are performed by the developer team. Hence, a question appears: to what developers the task of performing database schema changes should be assigned? the choice of the strategy to adopt may be crucial since it influences the evolution of the database schema. Different strategies can be adopted like (1) permitting each developer of the project to modify the database structures or (2) distributing responsibility among very few developers. Each strategy has its own advantages and weaknesses.

Permitting every developer. This strategy is the most *permissive* since every developer implied in the development project is allowed to modify the database structures at will. This strategy reduces the chances to have *database specialists* on who the the project success depends. This strategy can be a useful asset to struggle with the nowadays increasing turnover in the development teams. However, this strategy can unfortunately lead to a progressive deterioration of the database schema and to an increasing unmaintainability of the documentation over time. In addition, adopting such a strategy also necessitates a better synchronization and communication between developers. As illustration, the OSCAR system (as depicted in Figure 3.21) seems to adopt a quite permissive strategy; the distribution is quite homogeneous (no specialists) and large (> 40 developers). However, some problems of communication between developers were observed. Figure 3.22 depicts a typical scenario that one can observe when analyzing the evolution history of the OSCAR database. In February 2013, the `PROGRAM_ID` column was deleted by `anniezhou31@gmail.com`. Yet, all other developers do not seem to be aware of this deletion; indeed, on the same day, `marc@mdumontier.com` immediately recreated the column and finally realized, a few days later, that this deletion was actually intentional and definitely dropped it. This lack of communication is observable in numerous other examples.



History: vacancy_template[PROGRAM_ID]

Dates	Event	Committer
2012-04-29(13h35m09s)	CREATION	matthew.ma20110628@gmail.c
2013-02-15(23h13m22s)	DELETION	anniezhou91@gmail.com
2013-02-15(23h58m47s)	REAPPEARANCE	marc@mdumontier.com
2013-02-19(00h33m07s)	DELETION	marc@mdumontier.com

Figure 3.22: Illustration of misunderstandings between OSCAR developers while modifying the database structures.

Distributing among very few developers. The opposite strategy could consist in restricting the right to modify the database schema to a few developers. As consequence, those *privileged* developers become the *database specialists* and have all the database knowledge. Choosing such a *restrictive* strategy helps avoid disorder introduced by too many developers modifying the database schema. Moreover, limiting the number of developers in charge of modifying the database schema can facilitate keeping a precise and up-to-date database documentation. However, the disadvantage of this strategy is that if those specialists quit the project, its future developments will be seriously compromised. As illustration, it would be difficult to imagine the consequences if the OpenMRS database specialists quit the project (more than 66% of the database schema have been impacted by two database specialists).

Towards a balanced strategy? Whereas this dissertation does not aim to answer the question *"which strategy is better than the other one?"* (the answer would be probably different according to the context and policy of the project), we discuss here the weaknesses and advantages of each strategy. Our intuition would encourage a more *balanced* solution, namely homogeneously distributing the modification right among a limited but sufficient number of developers. An illustrative example is the distribution among the Broadleaf developers; the database knowledge is fairly distributed among less than 20 developers. The limited number allows keeping a certain discipline and order, while the homogeneous distribution limits the consequences of a developer leaving the project.

3.5 An In-Depth Case Study

In this Section, we particularly focus on a large scale case study of reverse engineering a complex information system, namely the OSCAR system. Through this case study, we motivate the benefits of using our approach in the context of a real-world project. In particular, we show how our approach facilitates at understanding the OSCAR system, with the objective of evolving the latter to fit new requirements.

3.5.1 Context: The OSCAR system

OSCAR (*Open Source Clinical Application Resource*) is full-featured Electronic Medical Record software system for primary care clinics. It has been under development since 2001 and is widely used in hundreds of clinics across Canada. As an open source project, OSCAR has a broad and active community of users and developers. The Department of Family Practice at McMaster University, which has managed OSCAR development efforts from inception to 2012, has recently transferred oversight of ongoing development to a newly formed not-for-profit company called OSCAR-EMR. This move was motivated by a new regulatory requirement to undergo ISO certification (*ISO 13485 Medical devices – Quality management systems*).

OSCAR Architecture. OSCAR relies on a Web application architecture following the classical 3-tier paradigm. It employs a Java-based technology stack, making use of Java Server Pages (JSP), Enterprise Java Beans (J2EE) and several frameworks such as Spring, Struts and Hibernate. The source code comprises approximately two million lines of code with a rough distribution of 600 kLOC for the application logic, 1200 kLOC for the presentation layer and 100 kLOC for the persistence layer. OSCAR uses MySQL as the relational database engine and a combination of different ways to access it, including Hibernate object-relational middleware, Java Persistence Architecture (JPA) and dynamic SQL (via JDBC). The reason for this combination of technologies is the constant and ongoing evolution history of the product, which originated from JDCB, via Hibernate to JPA.

OSCAR database. The OSCAR database schema has over 440 tables and many thousands of attributes. At the time of conducting our study, the database schema of the OSCAR distribution did not contain any information on relationships between tables (foreign keys) and no documentation was available about the schema. We later learned that the missing relationships were due to the evolution history of OSCAR, which has been using the older MyISAM database engine provided by MySQL, which does not support foreign keys. A port to the newer InnoDB engine is underway, which will eventually allow foreign keys to be defined explicitly.

OSCAR software repositories. The OSCAR community utilizes a range of software repositories and tools, including a feature request and bug tracking system (provided by Sourceforge), a source code submission and review system (Gerrit Code Review), a git-based configuration management system, a community Wiki (based on Plone) and three active mailing lists (one for developers and two for users of different levels of technical expertise).

The need to understand the database schema. The OSCAR database has grown organically over many years and knowledge about its internal structure is distributed among very few developers who have been contributing to specific functions of the system (e.g., prescription writer, representation of lab results etc.). Our need to understand the OSCAR database schema originated from our collaboration with the SCOOP team[8] involved in a project with the goal to develop software for a primary care research network (PCRN). The purpose of the PCRN is to integrate health information kept in primary care EMR software in order to make them accessible to medical research and data mining. An important step in developing the PCRN software is to create "export conduits" for transferring health data from the EMR into a research database for subsequent query processing. Due to its popularity (second largest market share in British Columbia) and openness, OSCAR has been chosen as one of the first EMR products to interface with the emerging PCRN.

While designing early versions of the PCRN data migration adapter for OSCAR, the SCOOP team found that they were running into questions pertaining to the

[8]http://scoop.leadlab.ca/

database schema. Of course, as could be expected for any heavily evolved, real-world system, some of them had to do with the fact that the database schema lacked documentation. Moreover, the schema did not contain any declared relationships (foreign keys). Other questions were of a more semantic and puzzling nature. For example, when attempting to design the function to export data on patient immunization records, they found two seemingly unrelated schema structures covering the same semantic issue. One schema structure revolved around tables entitled "*immunizations*" and "*configimmunization*" while the other schema structure revolved around tables entitled "*preventions*" and "*preventionsext*". During this project they found that taking into consideration the evolution history of the database schema was helpful in answering questions like these (we found out that the "*preventions*" structured superseded the "*immunizations*" structure but has still be retained in order to deal with legacy data). This motivated us to investigate more formally OSCAR's evolution history and develop methods and tools to help with this investigation.

3.5.2 Report

We analyzed the history of the OSCAR database schema over a period of almost ten years (22/07/2003 - 27/06/2013). During this period, a total of 670 different schema versions can be found in the project's GitHub repository. The earliest schema version analyzed (22/07/2003) includes 88 tables, while the latest schema version considered (27/06/2013) comprises 445 tables.

Based on those 670 schema versions, we computed the corresponding historical schema. The historical schema contains 552 tables and 18,443 columns. Figures 3.10 and 3.15 depict the historical schema as visualized, respectively, in 2D and 3D. In both representations, we applied our colourization procedure depending on its age and its liveness (see Section 3.3.6).

We then made use of DAHLIA to query the historical schema and extract interesting statistics regarding the evolution of the schema. Some of those statistics for the OSCAR database are given below.

Figure 3.23 (A) depicts the evolution of the number of tables in the OSCAR schema. We can observe that this number keeps increasing. Indeed, we found out that in our case study, developers are very reluctant in removing tables in order to achieve backward compatibility and avoid the important impact of a schema refactoring on the data and the application programs. From the same figure, we can also easily identify those schema versions that could be considered as "major releases", i.e., those versions where an important number of tables have been added and/or deleted.

Figure 3.23 (B) represents the evolution of the total number of columns in the OSCAR schema, that has grown from 2,443 to 13,364 columns in circa ten years. Fortunately, this number follows a similar trend as the evolution of tables, keeping the average number of columns per table quite stable over time (around 25).

Figure 3.24 (A) provides some finer-grained information about the creation and deletion of tables. One can easily notice that OSCAR tables are rarely removed. The evolution of the schema consists (most of the time) of adding tables, while not

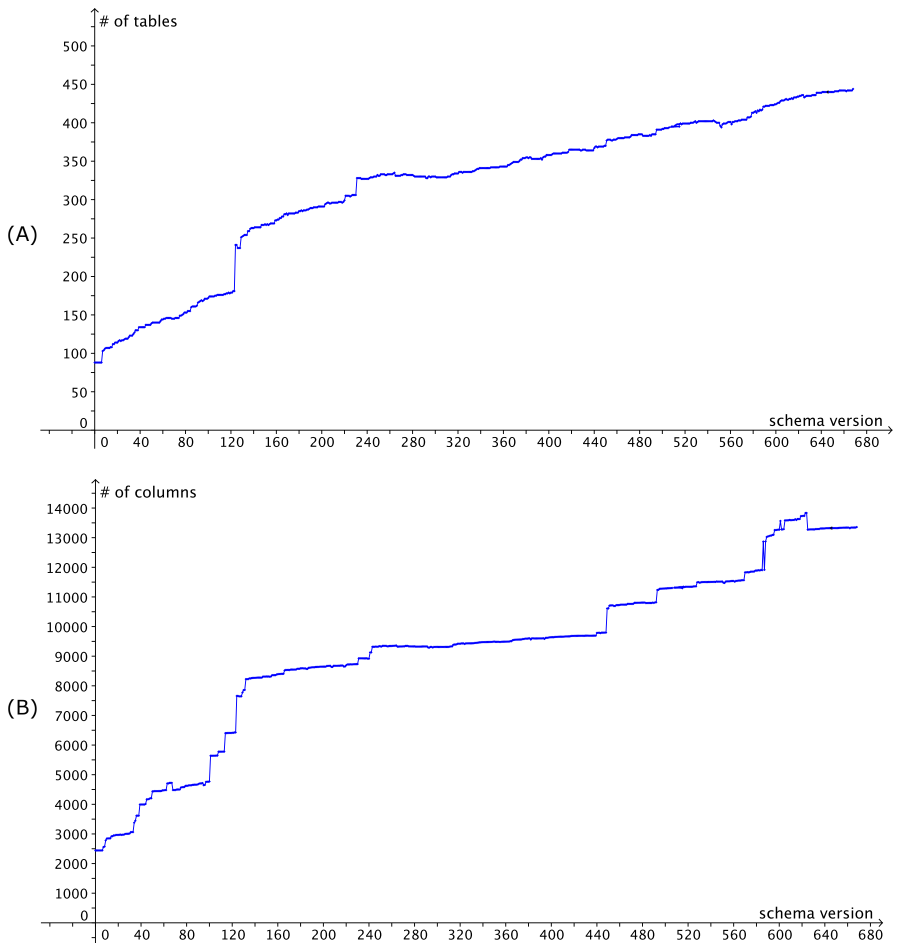


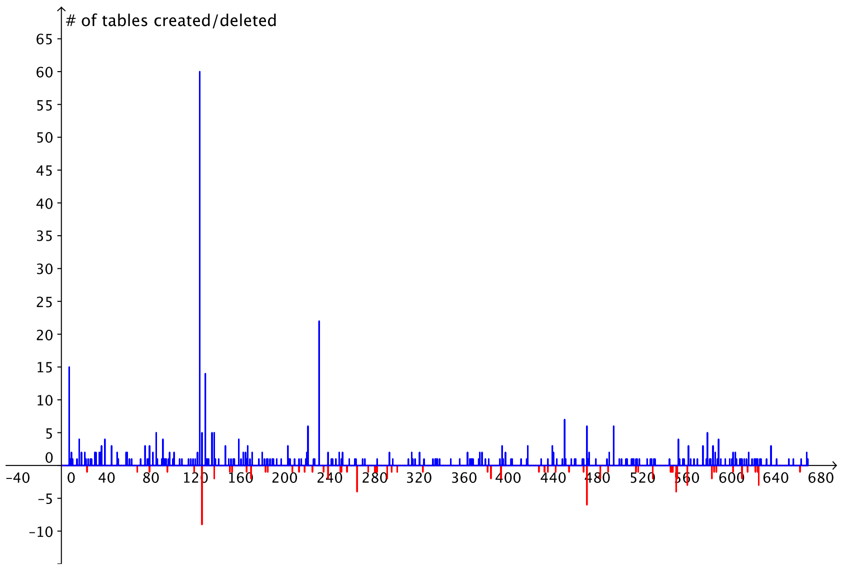
Figure 3.23: Extracted information from the OSCAR historical schema: (A) evolution of the number of tables; (B) evolution of the number of columns.

replacing or splitting them up. The total number of deleted tables is around 30, and we can again quickly identify the major release time periods. The observation is similar for the ratio of created and deleted columns, as shown in Figure 3.24 (B). The number of column creations is, indeed, often greater than the number of column deletions. During our test period, a total of 3,872 columns were removed, while 14,793 columns were created.

Table 3.4 provides statistics about the different types of schema changes applied to the OSCAR schema during the studied period. In those statistics, the *add column* operation corresponds to the creation of a new column in an existing table. Creating a table including *n* columns only counts as one *add table* operation, not as *n add column* operations.

Figure 3.25 (A) shows the classification of the OSCAR tables according to two

(A)



(B)

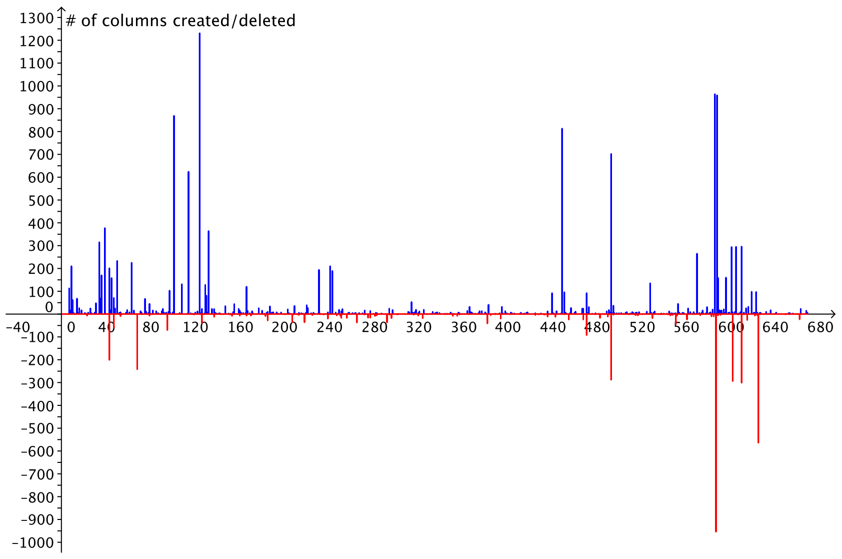
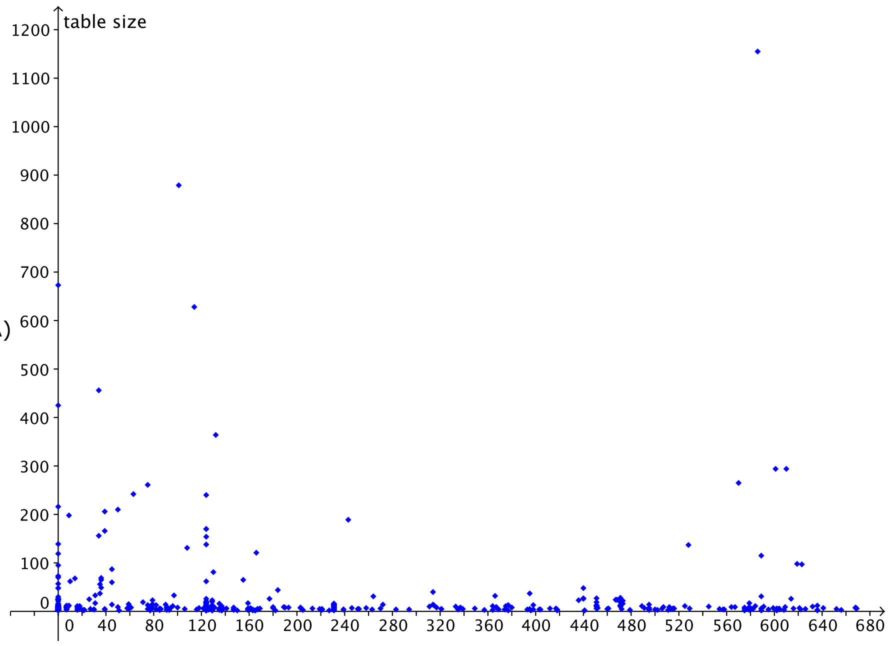


Figure 3.24: Extracted information from the OSCAR historical schema: (A) creation/deletion of tables; (B) creation/deletion of columns.

(A)


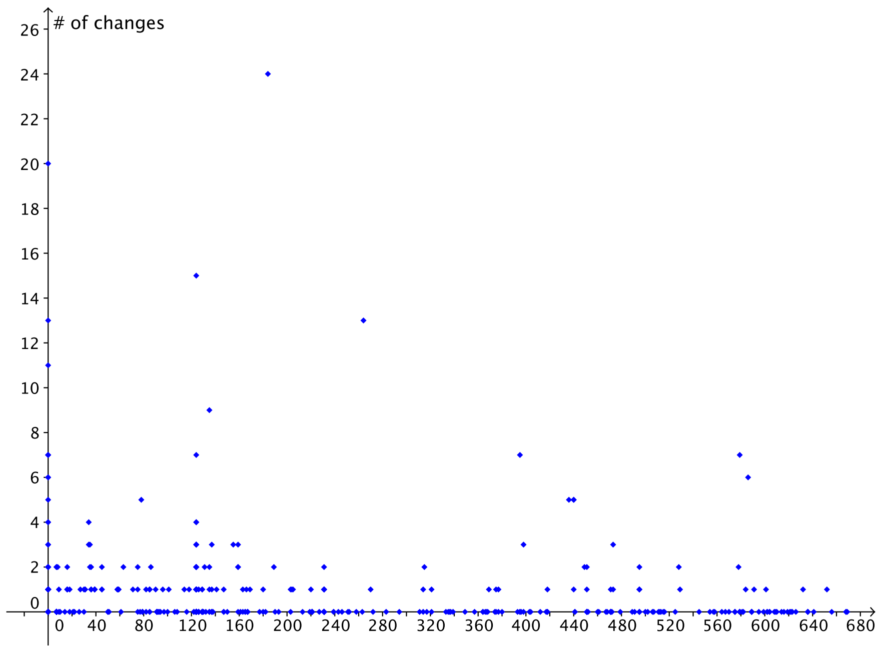
(B)


Figure 3.25: Extracted information from the OSCAR historical schema: (A) table creation version VS table size; (B) table creation version VS number of changes.

add table	443	add identifier	48
drop table	86	drop identifier	21
add column	2 091	add foreign key	3
drop column	703	drop foreign key	12
add index	125	change default value	23
drop index	32	change column type	342
add default value	1 509	set nullable column	32
drop default value	47	set non-nullable column	27

Table 3.4: Distribution of the OSCAR schema changes during the considered time period.

dimensions: their creation schema version (x-axis) and their size, expressed in number of columns (y-axis). One can notice that the large tables (over 200 columns) are created throughout the whole life of the system. This means that those tables are not a reflection of early design problems but are still being generated and used now. We investigated further this unexpected observation and found that the large tables were related to a user-programmable extension mechanism of OSCAR, which provides “power-users” with tools to add so-called *e-forms* to OSCAR for capturing form-based clinical data input.

Another interesting property, particularly in the context of database migration, is the *stability* of the tables. A table that has been created a long time ago, and that was not subject to frequent modifications can be considered *stable*. In Figure 3.25 (B), we characterize each table with respect to the number of times it has been modified since its creation, and we relate this information to its creation version. One can see that the database schema is globally stable, most of the tables having less than 4 modifications. As expected, it is mainly the oldest tables that have the highest number of changes, but there are a few exceptions.

Figure 3.26 relates the OSCAR developers with the evolution of the schema. Figure 3.26 (A) shows, for each developer, the number of distinct tables in the evolution of which he/she has been involved (by creating, updating or deleting the table). We observe than the few most active schema committers have hardly touched 20% of the OSCAR tables.

Figure 3.26 (B) provides a set of points (x, y) meaning that developer x has been involved in the evolution of table y. Such information is useful, for instance, to identify the **experts** of a given table, or the creator of a given schema object.

3.5.3 Discussion

Analyzing the evolution history of the OSCAR database schema has helped us to understand the current schema structure. We were surprised to see that schema structures are rarely deleted even if they are semantically replaced by others, for example as in the case of the “preventions” table structure replacing the “immunizations” structure. In these kinds of situations, our tool-based method allowed us to easily determine which schema structure superseded which other schema struc-

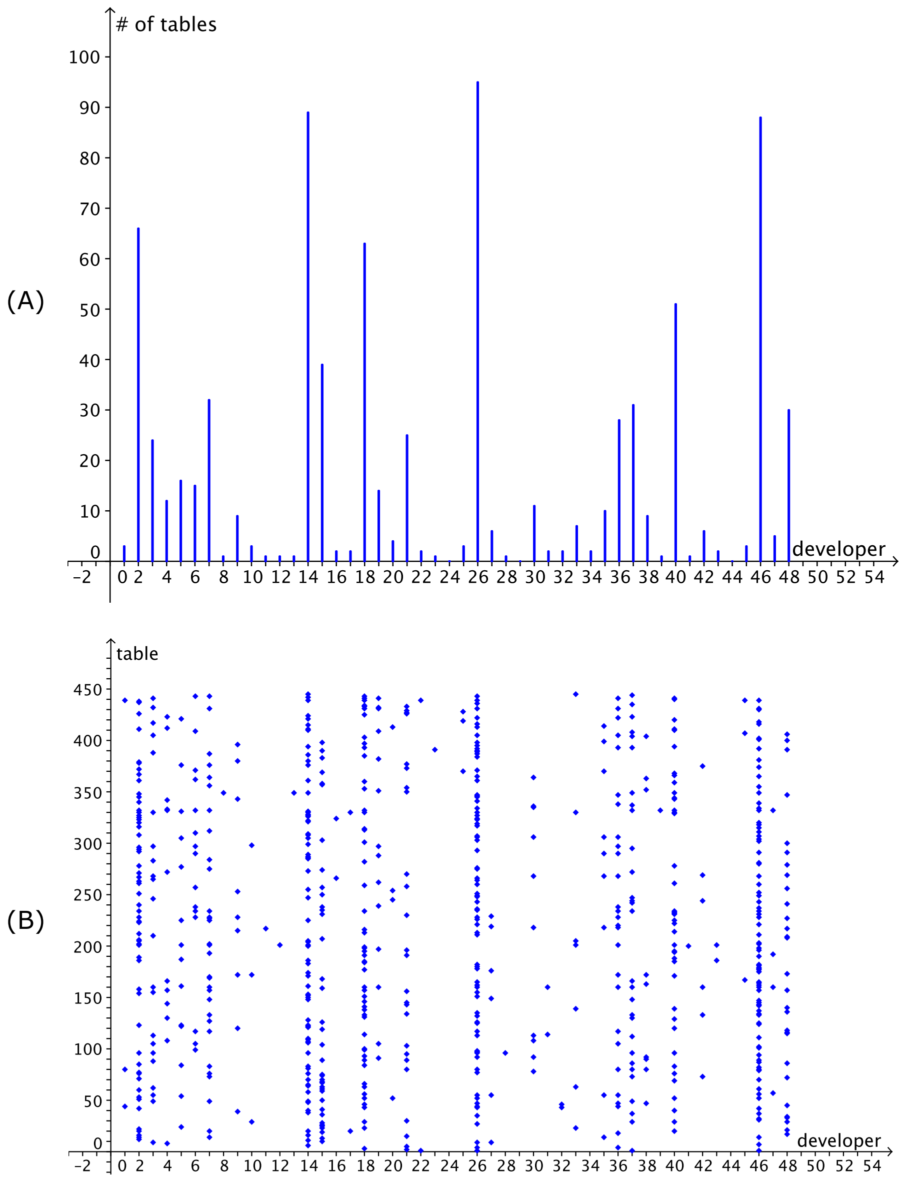


Figure 3.26: Information about the developers involved in the evolution of the OSCAR database schema. (A) number of tables impacted by each developer; (B) table vs developer matrix.

ture. There are potentially multiple explanations for keeping these legacy schema structures. The most likely explanation is domain-specific, namely that they are kept for medico-legal reasons: a patient record (electronic or otherwise) is a legal document that can only be amended but not deleted or arbitrarily modified. Keeping superseded database schema structures is the easiest way of accommodating older patient data that uses these outdated fields. Indeed, in case of the preventions tables, the OSCAR preventions user dialogue has at the bottom a link entitled "old immunizations", which allows the user to access the legacy data.

To some degree, the creation of new schema structures to semantically replace existing ones without removing them is similar to the well-known process of "cloning" in program code. Therefore, the preventions tables and the immunization tables could be considered as **database clones**. However, subtle differences exist to the concept of program clones. For example, program clones are usually still made up of functional code (as opposed to dead code), while the superseded database clone may indeed be considered as a "dead schema" from the point of view of a newer installation of the system that does not have to deal with legacy data.

Another important insight created by our schema evolution analysis method was a better understanding of the role of the many large-scale tables that contain hundreds or even thousands of attributes. The information content of these tables overlaps significantly with other parts of the transactional database. Therefore, our initial hypothesis (before considering the evolution history) was that these tables were relicts of early database designs. The evolution analysis, however, refuted this hypothesis and indicated that these tables are indeed being generated throughout the system lifetime. Further investigation showed that they were connected to an extension mechanism. The SCOOP team was therefore able to exclude them for the first phase of their database migration project, which simplified their task significantly.

3.6 Concluding Remarks

In this Chapter, we presented a historical approach that allowed us to analyze the evolution history of a given database schema. The method is based on the automated derivation of a historical schema, that includes all the schema objects involved in the entire lifetime of the database, each annotated with historical and temporal information. This historical analysis permits the detection of several schema changes occurred in the system lifetime. However, more sophisticated analysis techniques have to be developed in order to identify schema changes as splitting a table, or merging two tables into one single table. Since our approach only considers successive schema versions as input, such refactoring operations are currently seen as the combination of deletion and creation operations.

Secondly, we presented DAHLIA, a novel visualization tool that allows us to exploit and visualize a historical schema and its information. This tool proposes a 2D and 3D visualization mode. The latter implements the 3D city metaphor proposed by *CodeCity*.

Finally, we applied our whole tool-supported approach to several subject systems. In particular, we reported on our experiences made in the context of large-scale project aiming at evolving OSCAR, a large and complex medical information system. We showed the benefits of our approach.

Roadmap

In this chapter, we have presented a tool-supported method for analyzing the evolution history of legacy databases, and we have shown its benefits on a large scale case study of reverse engineering a complex information system.

Later, in Chapter 8, we show how analyzing the evolution history of NoSQL systems can be indispensable to prevent errors and data losses for future developments.

The next chapter (Chapter 4) addresses the problem of DISS evolution from another side; it presents an analysis approach to automatically detect and extract database access from the source code. In particular, this approach handles dynamic SQL and ORM technologies.

CHAPTER 4

STATIC ANALYSIS OF DYNAMIC DATABASE USAGE IN JAVA SYSTEMS

While the previous chapter focused on the exploitation of the database schema evolution history as main source of information to aid database schema reverse engineering, this chapter[a] *addresses the problem of data-intensive system evolution from another side. In this chapter, we present an analysis approach that aims to statically detect database accesses within the source and to recover their actual SQL values. This automatic approach, specifically designed for Java systems, targets three main database access technologies, i.e., JDBC, Hibernate and JPA. We secondly evaluate our approach on three real-life systems.*

[a]This chapter is an extension of our main conference paper [Meurice et al., 2016c] published in the proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAISE 2016).

4.1 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

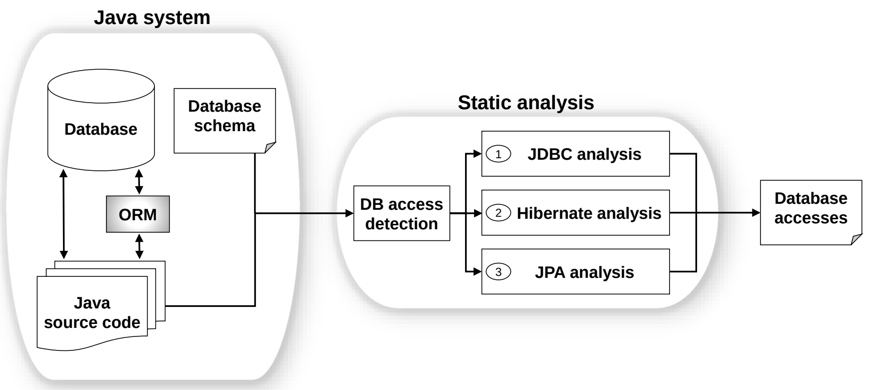


Figure 4.1: Overview of the proposed approach.

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 [TIOBE Programming Community Index, 2017], database manipulation has become increasingly complex in recent years. Indeed, a large-scale empirical study, carried out by Goeminne et al. [Goeminne and Mens, 2015], reveals that a wide range of dynamic database access technologies are used by Java systems to manipulate their database. 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 generated at runtime, through string concatenations and method calls. Moreover, some access mechanisms (like ORM frameworks) partly or fully hide the actual SQL queries executed by the programs [Cleve et al., 2010]. Those queries are generated at runtime before they are sent to the database server. This trend significantly complicates the task of identifying which portion of the source code accesses which portion of the database.

In this Chapter, 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 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 intraprocedural control-flow of the methods. It considers three of the most popular database access technologies used in Java systems, according to [Goeminne and Mens, 2015], namely JDBC, Hibernate, and JPA.

4.2 Approach

Figure 4.1 presents the overview of our approach which combines three different analyses: the JDBC, Hibernate and JPA analyses. The output of the full process is

```java
 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.executeQuery(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   }
21 }
```

Listing 4.1: Java code fragment using the JDBC API to execute a SQL query (line 10).

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.

4.2.1 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 Listing 4.1 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 interprocedural data-flow 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 analysis approach 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).

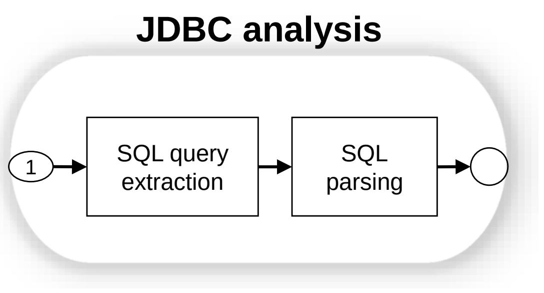


Figure 4.2: Overview of the JDBC analysis.

4.2.2 JDBC Analysis

Our JDBC analysis, depicted in Figure 4.2, focuses on the database accesses using the JDBC API, where we follow a two-phase process. A JDBC access recovery can be seen as a String expression recovery. Once our analysis approach has detected a JDBC access, it will then rebuild the corresponding SQL query. Finally, the SQL parsing process constructs the abstract syntax tree of the SQL query and identifies which part of the database schema is involved in that access; that is, it identifies the database tables and columns accessed with it. This identification relies on the actual database schema.

SQL Query Extraction. Algorithm 2 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 3). First, we *locally* resolve the expression and then we deal with the call graph extraction, when it is necessary. Let us apply Algorithm 2 on the sample code in Listing 4.1. 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.

(1) 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 $\in$ 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 $\in$ callGraph do
                //Extracting the input values from the current call
10              Expr[] inputExprs = extractParamValues(method, call, inputs)
11              Expr[][] inputValues = initialize()
12              for inputExpr $\in$ 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 $\in$ product do
17                  result.add(e)
18              end
19          end
20      end
21  end
22  return result
```

Algorithm 2: Algorithm for recovering the string values of a given Java expression.

(2) 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`'.

```
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 3: Description of the procedures used in Algorithm 2

SQL Parsing. In the end of the process, the SQL parsing phase will generate an AST for each extracted query and then extract the set of accessed objects. Figure 4.3 shows the corresponding AST constructed by parsing the query `select * from Provider order by provider_id`. The `Provider` table and its `provider_id` column are detected as the accessed objects. The analysis of the `Provider.*` field

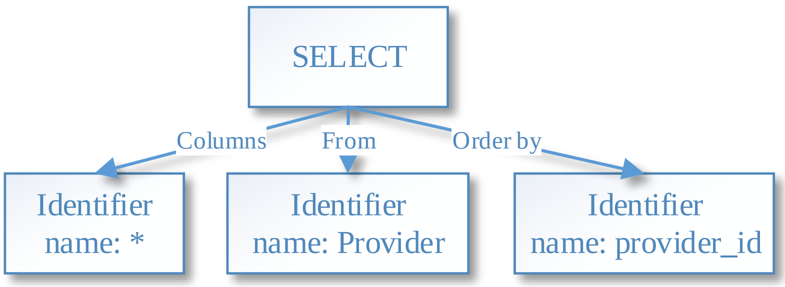


Figure 4.3: The corresponding AST constructed by parsing the query `select * from Provider order by provider_id`.

```
Book getBook() {
  String code = System.console().readLine(); //getting user input through the console
  String sql = "select * from book where code = " + code;
  rs = st.executeQuery(sql);
  ...
}
```

Listing 4.2: Example of SQL query construction using an input value given by the application user.

will be resolved as the set of columns of the `Provider` table. During the extraction of the accessed objects, our approach distinguishes the *explicitly* and *implicitly* accessed columns. In our example, the `provider_id` column is explicitly accessed, while every other column of the `Provider` table is implicitly accessed. The importance of this distinction will be further discussed and illustrated in Chapter 5, when supporting program-database co-evolution.

The success of the SQL query extraction process relies on the resolution of the values of string variables, for which we trace back string variables following the control flow and call graph of the application. When a variable cannot be resolved statically, the query cannot be fully extracted and contains *unresolved query fragments*. However, we design our SQL parsing phase to be robust and able to handle unresolved query fragments. For this step, it simply means that we use a special string '`@@null@@`' as a placeholder for such fragments. Listing 4.2 shows a typical method that constructs a SQL query to get a book in a database according to a code given by the application user. Since the code value introduced by the user through the console cannot be *statically* resolved, '`select * from book where code = @@null@@`' is the query string that we can extract from this method. Figure 4.4 shows the corresponding AST constructed by parsing the query.

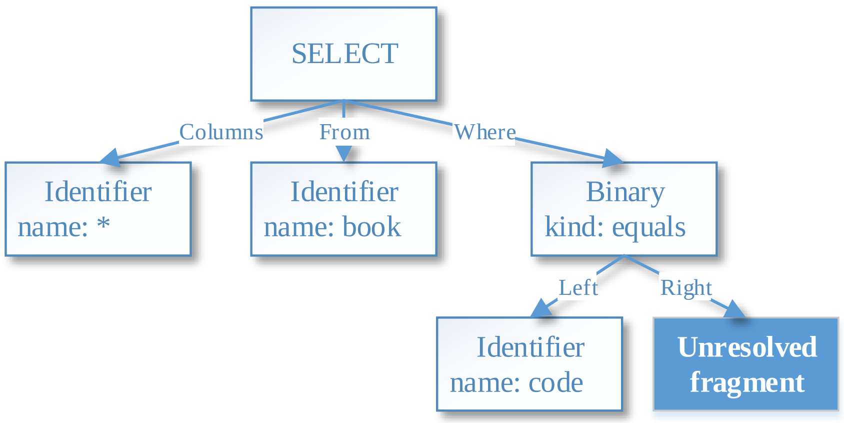


Figure 4.4: Example of AST containing an unresolved query fragment.

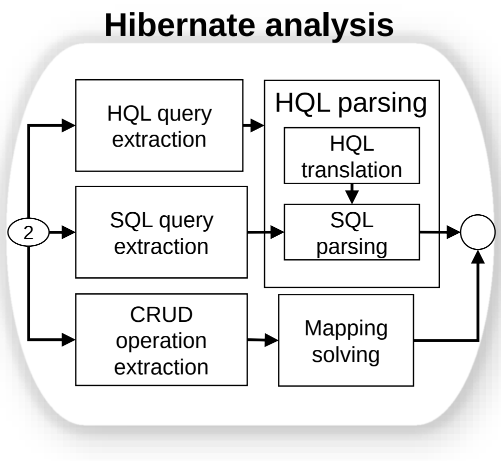
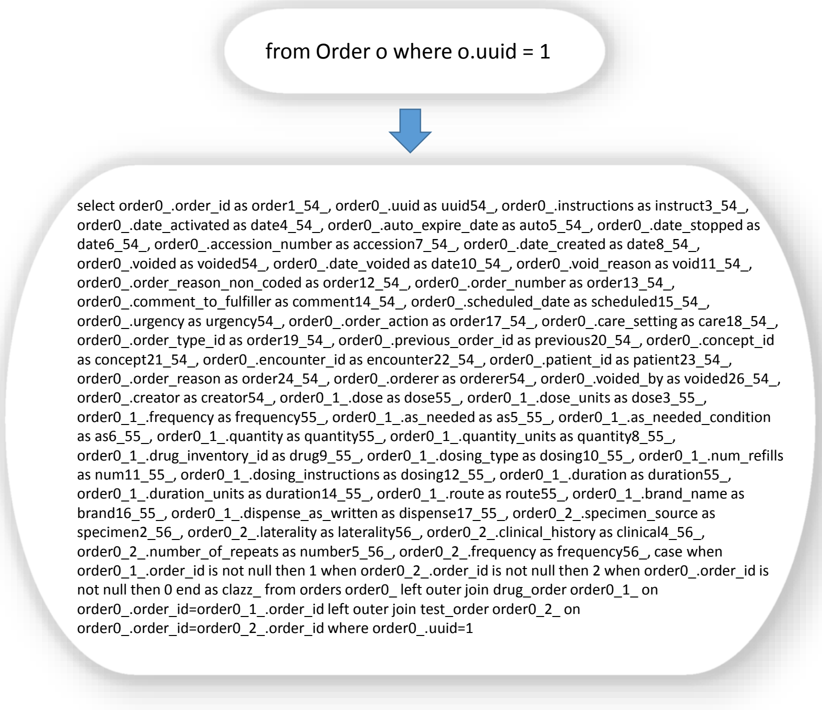


Figure 4.5: Overview of the Hibernate analysis.

4.2.3 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. Figure 4.5 depicts the overview of our Hibernate analysis.

Like the JDBC API, Hibernate also proposes different Java methods to execute either native SQL queries or HQL queries. The extraction process of those queries is similar to the JDBC analysis process (Algorithm 2). However, the HQL parsing



```
from Order o where o.uuid = 1
```

```
select order0_.order_id as order1_54_, order0_.uuid as uuid54_, order0_.instructions as instruct3_54_,
order0_.date_activated as date4_54_, order0_.auto_expire_date as auto5_54_, order0_.date_stopped as
date6_54_, order0_.accession_number as accession7_54_, order0_.date_created as date8_54_,
order0_.voided as voided54_, order0_.date_voided as date10_54_, order0_.void_reason as void11_54_,
order0_.order_reason_non_coded as order12_54_, order0_.order_number as order13_54_,
order0_.comment_to_fulfiller as comment14_54_, order0_.scheduled_date as scheduled15_54_,
order0_.urgency as urgency54_, order0_.order_action as order17_54_, order0_.care_setting as care18_54_,
order0_.order_type_id as order19_54_, order0_.previous_order_id as previous20_54_, order0_.concept_id
as concept21_54_, order0_.encounter_id as encounter22_54_, order0_.patient_id as patient23_54_,
order0_.order_reason as order24_54_, order0_.orderer as orderer54_, order0_.voided_by as voided26_54_,
order0_.creator as creator54_, order0_1_.dose as dose55_, order0_1_.dose_units as dose3_55_,
order0_1_.frequency as frequency55_, order0_1_.as_needed as as5_55_, order0_1_.as_needed_condition
as as6_55_, order0_1_.quantity as quantity55_, order0_1_.quantity_units as quantity8_55_,
order0_1_.drug_inventory_id as drug9_55_, order0_1_.dosing_type as dosing10_55_, order0_1_.num_refills
as num11_55_, order0_1_.dosing_instructions as dosing12_55_, order0_1_.duration as duration55_,
order0_1_.duration_units as duration14_55_, order0_1_.route as route55_, order0_1_.brand_name as
brand16_55_, order0_1_.dispense_as_written as dispense17_55_, order0_2_.specimen_source as
specimen2_56_, order0_2_.laterality as laterality56_, order0_2_.clinical_history as clinical4_56_,
order0_2_.number_of_repeats as number5_56_, order0_2_.frequency as frequency56_, case when
order0_1_.order_id is not null then 1 when order0_2_.order_id is not null then 2 when order0_.order_id is
not null then 0 end as clazz_ from orders order0_ left outer join drug_order order0_1_ on
order0_.order_id=order0_1_.order_id left outer join test_order order0_2_ on
order0_.order_id=order0_2_.order_id where order0_.uuid=1
```

Figure 4.6: Example of HQL query and its translated SQL form.

process is slightly different from the parsing phase of the JDBC analysis. Indeed, at this point we cannot just extract a SQL query string. Moreover, translating a HQL query into its corresponding SQL form is not trivial. Figure 4.6 shows an example of a HQL query and its translated SQL form. Therefore, we implemented a feature to be able to translate a 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., Listing 4.3. 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 2. Instead, we illustrate the use of that algorithm on Listing 4.3. The database access detection detects

```java
private Session session;
...
public void saveCustomer(Customer myCustomer) {
  saveObject(myCustomer);
}

public void saveObject(Object o) {
  session.save(o);
}
```

Listing 4.3: Hibernate operation on a mapped entity class instance. Insertion of a new customer.

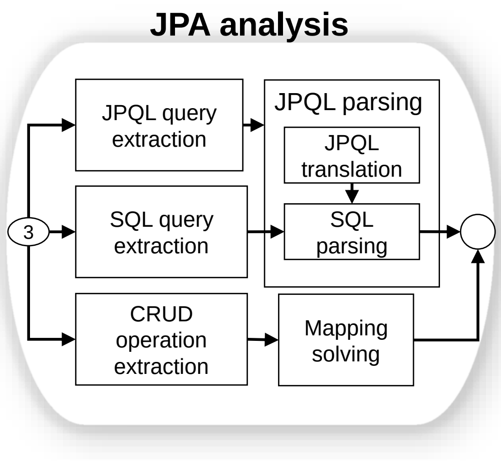


Figure 4.7: Overview of the JPA analysis.

`session.save(o)` as a database access. *o* is the expression to resolve and 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 mapping solving process will detect the mapping between the `Customer` class and its corresponding database table (e.g., table *customer*).

4.2.4 JPA Analysis

The JPA analysis, depicted in Figure 4.7, concentrates on the database accesses by means of JPA. Like Hibernate, JPA proposes Java methods to execute either native

```
EntityManagerFactory emf = ...;
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Order order = ...;
em.persist(order);
em.getTransaction().commit();
em.close();
```

Listing 4.4: JPA operation on a mapped entity class instance. Creation and insertion of a new order.

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 2 and then we parse the JPQL query. The JPQL parsing process 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., Listing 4.4. 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 mapping solving process will consider the JPA annotations which define this mapping.

4.2.5 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 `CheckDrugOrderUnit.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**]

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 4.1: Size metrics of the systems

4.3 Evaluation

In this Section, we evaluate our approach on three real-life systems, namely OSCAR[1], OpenMRS[2] and Broadleaf[3]. The detailed results of this evaluation are available as an online appendix[4].

4.3.1 Evaluation Environment

Table 4.1 presents an overview of the main characteristics of the target systems. OSCAR combines JDBC, Hibernate and JPA to access the database. OpenMRS uses a MySQL database accessed via Hibernate and dynamic SQL (JDBC). Broadleaf uses a relational database accessed via JPA.

Table 4.2 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 database accesses detected by our approach.

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

Table 4.2: Number of database accesses per technology

Figure 4.8 shows the set of tables and columns accessed 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 accesses). Concerning OpenMRS, the biggest database part is accessed by Hibernate (713 columns for 687 accesses) whereas JPA is the only used mechanism in Broadleaf (431 columns for 930 accesses).

Table 4.3 depicts, for each system, the number of recursive calls needed for completely recovering a code location accessing the database, i.e., the number of recursive calls in Algorithm 2. In OSCAR, one can notice that 4 recursive calls are

[1] Checked out from GitHub at August 26, 2014.
[2] Checked out from GitHub at November 6, 2014.
[3] Checked out from GitHub at February 20, 2015.
[4] https://staff.info.unamur.be/lme/CAISE16/

System	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

Table 4.3: Number of recursive calls in Algorithm 2 required to completely reconstruct database accesses.

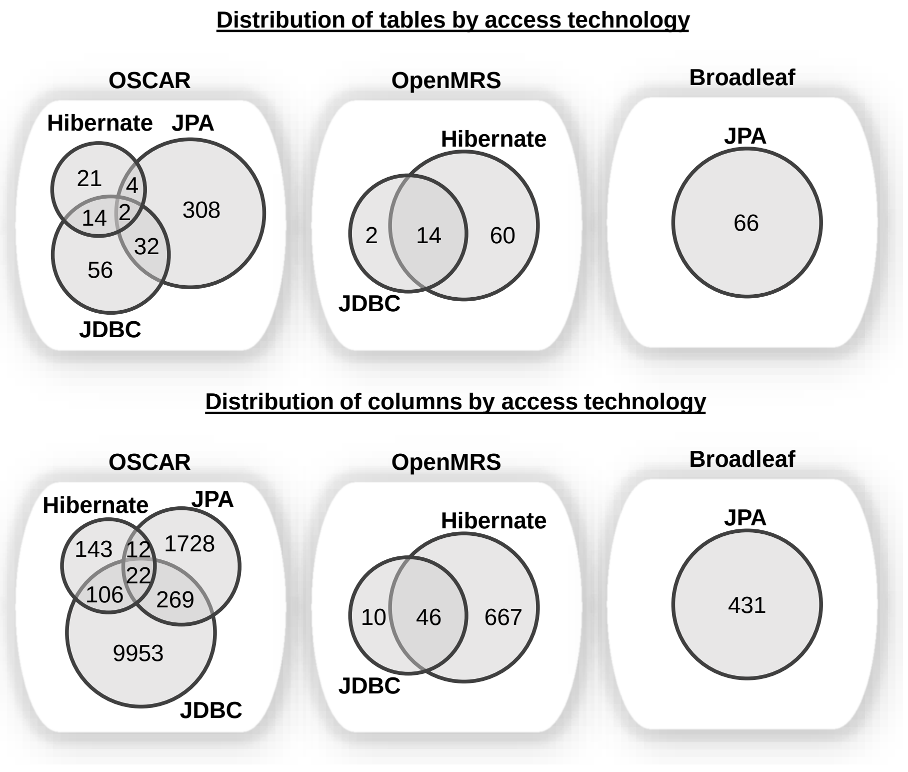


Figure 4.8: Distribution of *tables* and *columns* by access technology

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 4.3, we note that OSCAR is the most complex, recursive calls being often necessary to recover the database accesses. In contrast, one can observe that in OpenMRS and Broadleaf, most database accesses are built and executed within the same method.

4.3.2 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*[5] to collect trace logs containing the exact string values of all of the queries sent to the database and their corresponding stack traces.

System	Java Files	LOC	Number of Test Runs
OSCAR	361	49 086	1 311
OpenMRS	352	76 960	3 258
Broadleaf	109	17 633	255

Table 4.4: Unit tests of the systems under question.

System	Technology	Covered Classes	Covered Locations
OSCAR	JDBC	1.05%	0.28%
	Hibernate/JPA	65.00%	56.79%
OpenMRS	JDBC	16.00%	13.56%
	Hibernate/JPA	69.57%	58.16%
Broadleaf	JDBC		
	Hibernate/JPA	33.96%	19.10%

Table 4.5: Coverage values of the unit tests

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

Table 4.4 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 4.5 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, these coverage values are, respectively, 65% and 69.57% 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 4.2.5) of the extracted queries. Moreover, we compare the string values of the SQL queries.

Table 4.6 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

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 4.6: Percentage of successfully extracted 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 4.7: Percentage of valid queries for each system

is 99% (94 JPA accesses out of 95).

Percentage of valid queries. It is possible that we extract an incorrect query that will never be executed by the program. 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 4.3.3). We consider that these queries are *invalid* and represent 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 4.7. 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).

4.3.3 Limitations

As we have seen, our approach reached good results when applied to real-life large 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.

```
Session session = ...;
...
StringBuilder sb = new StringBuilder();
sb.append(" select distinct patient.id from DrugOrder where voided = false and patient.
    voided = false ");
if (drugList != null) {
  sb.append(" and drug.id in (:drugIdList) ");
}
if (startDateFrom != null && startDateTo != null) {
  sb.append(" and dateActivated between :startDateFrom and :startDateTo ");
}
...
Query query = session.createQuery(sb.toString());
```

Listing 4.5: HQL query construction by means of a StringBuilder object.

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 4.6), 49 are due to the use of StringBuilder objects for creating the query value. This obviously affects the percentage of successfully extracted queries. Listing 4.5 shows an encountered example of the use of StringBuilder to create a HQL query[6].

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. Listing 4.6 depicts an encountered example of invalid extracted HQL query due to dependent boolean conditions. From this example, four HQL queries were recovered:

(a) select concept from Concept as concept left join concept.names as names where names.conceptNameType = 'FULLY_SPECIFIED'

[6]http://bit.ly/1XNeL4e

(b) from Concept as concept
(c) select concept from Concept as concept
(d) from Concept as concept left join concept.names as names
where names.conceptNameType = 'FULLY_SPECIFIED'

However, the two last queries are invalid since the two boolean conditions (lines 3 and 6) are dependent[7].

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

Listing 4.6: HQL query construction - dependency between two if statements.

4.4 Concluding Remarks

In this Chapter, we presented a static program analysis technique allowing us to automatically detect and recover database accesses within the source code. This technique is specifically designed for Java systems (the most popular programming language today [TIOBE Programming Community Index, 2017]). This analysis approach permits the automatic recovery of SQL queries which are dynamically constructed in the code or which are partially/fully hidden because generated by the ORM layer. In particular, our approach targets three of the most popular Java access technologies, i.e., JDBC, Hibernate and JPA. We secondly conducted an evaluation of our approach based on three real-life large systems and reached good results. We also identified some limitations of our approach (mainly due to its static nature).

The automated detection and recovery of the database usage in the source code is a first essential step towards the implementation of an approach supporting the co-evolution between the programs and the database. Therefore, Chapter 5 is built on the use of our analysis approach.

Roadmap

In this chapter, we have presented a static analysis approach that automatically detects and recovers database accesses from the source code; this approach is specifically designed for Java systems and targets at three main database access technologies, i.e., JDBC, Hibernate and JPA.

In the next chapter (Chapter 5), we reuse this static analysis method to build a historical analysis approach that allows us to analyze, at a fine-grained level, how the program source code and the database schema have co-evolved over time.

[7] http://bit.ly/1Y0TJAT

Later, in Chapter 8, we present direct applications of this analysis approach to different fields.

CHAPTER 5

Analyzing the Evolution of Database Usage in Data-Intensive Software Systems

This chapter[a] *builds on the previous chapter. Based on the static analysis approach, we present a historical analysis approach that allows us to analyze, at a fine-grained level, how the program source code and the database schema have co-evolved over time. We then motivate the benefits of this historical approach on real-life systems.*

[a]An extended version of this chapter appeared in the book *Software Technology : 10 Years of Innovation*, published by John Wiley & Sons in 2016 [Meurice et al., 2016a].

5.1 Introduction

In previous chapters, we observed that the communication between the program and the database can be ensured by several kinds of database access technologies. For instance, ORM middleware provides programmers an external, object-oriented *view* on the database schema. Both schemas can evolve asynchronously, each at their own pace, often under the responsibility of independent teams. Therefore, severe inconsistencies may then progressively emerge due to undisciplined evolution processes. In addition, the high level of dynamicity of current database access technologies makes it hard for a programmer to figure out which SQL queries will

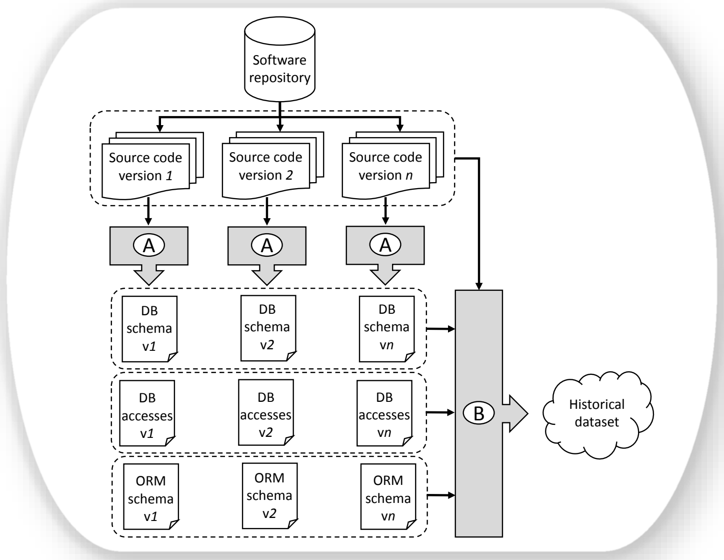


Figure 5.1: The overview of our approach. Step Ⓐ : the extraction; Step Ⓑ : the historization.

be executed at a given location of the program source code, or which source code methods actually access a given database table or column. Things may become even worse when multiple database access technologies co-exist within the same software system; co-evolving the database and the program may then require to master several different languages and technologies.

In this context, this Chapter proposes a historical approach allowing us to understand, at a fine-grained level, how systems evolve over time, how the database and code co-evolve and how several technologies may co-exist into the same system. In particular, our approach aims to compute a *historical database* gathering information about the evolution of several system artefacts, namely the source code, the database schema, the database usage and the ORM usage. Querying this historical database will allow us to understand how the program and database have co-evolved over time and how this evolution has led to the current system state.

5.2 Approach

The objective of our automatic approach is to compute a historical dataset gathering information about the evolution of several system artefacts, namely the source code, the database schema, the database usage and the ORM usage. Our approach, depicted in Figure 5.1, comprises two steps: the **extraction** and **historization** steps. The extraction step exploits the system's history (i.e., the versioning repository) in order to extract information about the evolution of (1) the database schema, (2) the database usage and (3) the ORM usage. Finally, the historization step *historizes*

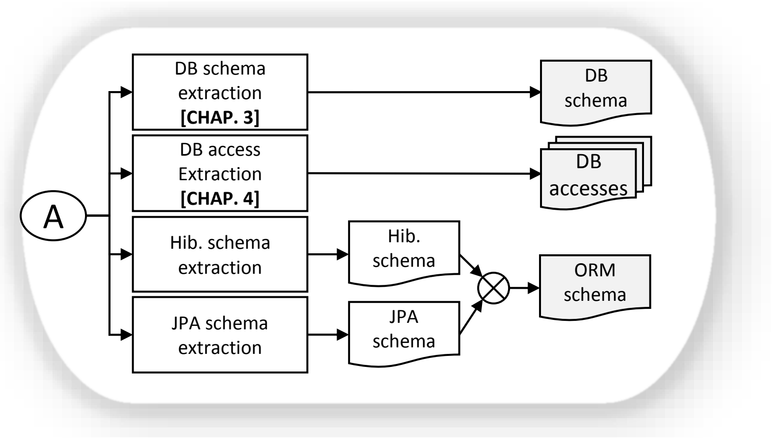


Figure 5.2: Phase Ⓐ of our approach depicted in Figure 5.1: the Extraction.

all this information to derive a historical dataset. Querying the resulting historical dataset will allow us to understand how the program and database schema co-evolve over time.

In particular, our approach focuses on the history 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 [TIOBE Programming Community Index, 2017]. Our approach currently focuses on three of the most popular Java technologies (according to [Goeminne and Mens, 2015]), namely JDBC, Hibernate, and JPA.

5.2.1 Extraction

The extraction phase, depicted in Figure 5.2, exploits the versioning system in order to separately analyze each successive system version. This analysis process is composed of 4 steps: the database schema extraction, the database access extraction, the Hibernate schema extraction and the JPA schema extraction.

Database Schema Extraction

This phase aims to extract the database schema at a given system version. We previously addressed this problem in Section 3.2.

Database Access Extraction

This phase aims to detect and recover the SQL queries and ORM accesses within the source code. We previously addressed this problem in Section 4.2.

As output of this step, we have a set of recovered database accesses and useful information about them, as described in Section 4.2.5; for each access, we know the accurate code location and the database tables and columns involved in it. The code

```
<!-- The document root. -->
<!ELEMENT hibernate-mapping (
  ...
  class*
)>

<!-- Root of an entity class hierarchy. Entities have their own tables. -->
<!ELEMENT class (
  (id|composite-id),
  discriminator?,
  (property|component)*,
  join*,
  subclass*,
  ...
)>
  <!ATTLIST class name CDATA #REQUIRED>
  <!ATTLIST class table CDATA #IMPLIED>
  <!ATTLIST class schema CDATA #IMPLIED>
```

Listing 5.1: DTD schema sample of any Hibernate configuration file.

```
<hibernate-mapping>
  <class name="Customer" table="customer" schema="finance" />
</hibernate-mapping>
```

Listing 5.2: Example of persistence class declaration in the Hibernate mapping document.

location of a given access is expressed by the minimal program path necessary for creating and executing the database access.

Hibernate Schema Extraction

Hibernate mappings are usually defined in an XML document. The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations and not table declarations. The mapping document is structured according to the DTD schema which is specified in the Hibernate documentation[1]. Listing 5.1 shows a fragment of this DTD schema.

The Hibernate schema extraction phase targets at analyzing this mapping document in order to automatically recover the defined mappings between source code instances and database elements. The objective is to detect the code locations where mappings are defined and to extract the *Hibernate schema*. The latter is made of tables and columns mapped to the code via Hibernate, and thus represents, at least, a sub-schema of the actual database schema.

Entity class. Developers can declare a persistent entity class mapped to a database table by using the `class` element. Listing 5.2 depicts an example of mapping between the `Customer` class and the `customer` table from the `finance` schema.

[1]The DTD schema can be found in its entire form at http://hibernate.org/dtd/.

```xml
<hibernate-mapping>
  <class name="Customer" table="customer" schema="finance">
    <property name="city" column="city" />
    <property name="country" column="country" />
  </class>
</hibernate-mapping>
```

Listing 5.3: Example of entity property declaration in the Hibernate mapping document.

```java
public class Customer {
  private String city;
  private String country;
  ...
}
```

Listing 5.4: Java class concerned by the mapping defined in Listing 5.3.

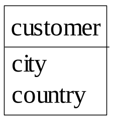


Figure 5.3: Database table concerned by the mapping defined in Listing 5.3 and 5.6.

Entity properties. The `property` element permits developers to declare a mapping between a class attribute and a database column. Listing 5.3 shows an example of mappings between the `customer.city` and `customer.country` columns, and respectively, the `Customer.city` and `Customer.country` attributes. Listing 5.4 and Figure 5.3 depict, respectively, the Java class and the database table that are mapped in this use case.

In addition, Hibernate permits to map a single entity to multiple tables. The `join` element allows developers to declare a *secondary* table in a mapping. In Listing 5.5, both `customer` and `address` tables are mapped to the `Customer` class. Both tables are joined by `address.customer_id`, the foreign key column. Listing 5.4 (remaining unchanged) and Figure 5.4 depict, respectively, the Java class and the database tables that are mapped in this use case.

Hibernate also allows the mapping between a class attribute and a group of database columns. Indeed, the `component` element maps properties of a child object to columns of the table of a parent class. Components can, in turn, declare their own properties and components. Listing 5.6 depicts an example of component; the `Address.city` and `Address.country` attributes are, respectively, mapped to the `customer.city` and `customer.country` columns. The `Customer.address` attribute is an instance of the `Address` class. Listing 5.7 and Figure 5.3 depict, respectively, the Java class and the database table that are mapped in this use case.

```
<hibernate-mapping>
  <class name="Customer" table="customer" schema="finance">
    <join table="address">
      <key column="customer_id" />
      <property name="city" column="city" />
      <property name="country" column="country" />
    </join>
  </class>
```

Listing 5.5: Example of entity join table declaration in the Hibernate mapping document.

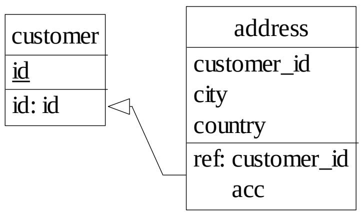


Figure 5.4: Database tables concerned by the mapping defined in Listing 5.5.

```
<hibernate-mapping>
  <class name="Customer" table="customer" schema="finance">
    <component name="address" class="Address">
      <property name="city" column="city" />
      <property name="country" column="country" />
    </component>
  </class>
</hibernate-mapping>
```

Listing 5.6: Example of entity component declaration in the Hibernate mapping document.

Entity identifier. Mapped classes must declare the primary key column of the database table. The `id` element defines the mapping from that property to the primary key column.

```java
public class Customer {
  private Address address;
  ...
}

public class Address {
  private String city;
  private String country;
}
```

Listing 5.7: Java class concerned by the mapping defined in Listing 5.6.

```xml
<hibernate-mapping>
  <class name="Customer" table="customer" schema="finance">
    <id name="customer_id" type="int" column="id" />
  </class>
</hibernate-mapping>
```

Listing 5.8: Example of entity identifier declaration in the Hibernate mapping document.

```java
public class Customer {
  private int customer_id;
  ...
}
```

Listing 5.9: Java class concerned by the mapping defined in Listing 5.8.

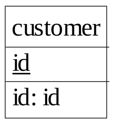


Figure 5.5: Database table concerned by the mapping defined in Listing 5.8.

Listing 5.8 depicts a mapping between the `customer.customer_id` primary key column and the `Customer.id` class attribute. Listing 5.9 and Figure 5.5 depict, respectively, the Java class and the database table that are mapped in this use case.

However, a table with a composite key can also be mapped with multiple properties of the class as identifier properties. The `composite-id` element accepts property mappings as child elements. Listing 5.10 depicts an example of composite identifier composed of the `firstName` and `lastName` columns, which are respectively mapped to the `first_name` and `last_name` class attributes. Listing 5.11 and Figure 5.6 depict, respectively, the Java classes and the database table that are mapped in this use case.

Entity inheritance. Polymorphic persistence is also allowed and requires the dec-

```xml
<hibernate-mapping>
   <class name="Customer" table="customer" schema="finance">
    <composite-id name="id" class="CustomerId">
      <property name="first_name" column="firstName" />
      <property name="last_name" column="lastName" />
    </composite-id>
  </class>
</hibernate-mapping>
```

Listing 5.10: Example of composite identifier declaration in the Hibernate mapping document.

```java
public class Customer {
  private CustomerId id;
  ...
}

public class CustomerId {
  private String first_name;
  private String last_name;
}
```

Listing 5.11: Java classes concerned by the mapping defined in Listing 5.10.

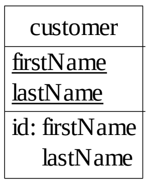


Figure 5.6: Database table concerned by the mapping defined in Listing 5.10.

```xml
<hibernate-mapping>
  <class name="Payment" table="PAYMENT">
    ...
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
      ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
      ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
      ...
    </subclass>
  </class>
</hibernate-mapping>
```

Listing 5.12: Example of entity inheritance declaration in the Hibernate mapping document.

laration of each subclass of the root persistent class. For instance, the `subclass` declaration is used to establish a *table-per-class-hierarchy* mapping strategy. Listing 5.12 illustrates an example of subclasses defined in Hibernate. The root class is mapped to the `PAYMENT` table while each of its subclass corresponds to a particular payment mode.

In addition, the `discriminator` element is required to declare a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. In our case, the `PAYMENT_TYPE` column is restricted to a set of predefined *discriminator* values which are used to determine the selected payment mode. Listing 5.13 and Figure 5.7 depict, respectively, the Java classes and the database table that are mapped in this use case.

```java
public class Payment{
  ...
}

public class CashPayment extends Payment {
  ...
}

public class ChequePayment extends Payment {
  ...
}
```

Listing 5.13: Java classes concerned by the mapping defined in Listing 5.12.

PAYMENT
PAYMENT_TYPE

Figure 5.7: Database table concerned by the mapping defined in Listing 5.12.

As output of the Hibernate schema extraction, we obtain a set of tables and columns concerned by Hibernate mappings; for each table/column, we know the code location(s) where a mapping involving the given object is declared.

JPA Schema Extraction

JPA offers similar mechanisms to establish mappings between database elements and code instances. The mappings are defined through annotations instead of `hbm.xml` files. Those annotations, which are specified in the JPA documentation[2], are declared in the source code itself.

The JPA schema extraction aims at analyzing the JPA annotations in order to automatically detect and recover the database-code mappings. Similarly to the Hibernate schema extraction, the objective is to detect code locations where JPA mappings are defined and to extract the JPA schema. The latter represents, at least, a sub-schema of the actual database schema.

Entity class. Every persistent entity class is declared using the `@Entity` annotation, at the class level. In addition, the `@Table` annotation can be used to determine the mapped table. Listing 5.14 illustrates the declaration of the mapping between the `customer` table and the `Customer` class, through JPA annotations.

Entity properties. Developers can declare entity properties, i.e., mappings between a database column and a class attribute, by use of the `@Column` annotation. Listing 5.15 shows an example of entity property declaration. Figure 5.3 depicts the table concerned by this use case.

JPA also permits to map a single entity to multiple tables. A secondary table can be declared by use of the `@SecondaryTable` annotation. In Listing 5.16, both

[2]The JPA specifications can be found at http://www.oracle.com/technetwork/java/javaee/documentation/index.html.

```java
@Entity
@Table(name="customer")
public class Customer {
  ...
}
```

Listing 5.14: Example of persistent class declaration through JPA annotation.

```java
@Entity
@Table(name="customer")
public class Customer {
  @Column
  private String city;

  @Column
  private String country;
  ...
}
```

Listing 5.15: Example of properties declaration through JPA annotation.

```java
@Entity
@Table(name = "customer")
@SecondaryTable(name = "address", pkJoinColumns = @PrimaryKeyJoinColumn(name = "
      customer_id"))
public class Customer {
  ...
  @Column(table = "address")
  private String city;

  @Column(table = "address")
  private String country;
  ...
}
```

Listing 5.16: Example of secondary table declaration through JPA annotation.

`customer` and `address` tables are mapped to the `Customer` class. Both tables are joined by `address.customer_id`, the foreign key column. Figure 5.4 depicts the tables concerned by this use case.

It is also possible to declare an embedded component inside an entity. Component classes have to be annotated with the `@Embeddable` annotation. Listing 5.17 shows an embedded component declaration. Figure 5.3 depicts the table concerned by this use case.

Entity identifier. The `@Id` annotation declares the identifier property of an entity. Listing 5.18 depicts the identifier declaration within the `Customer` persistent class. Figure 5.5 depicts the table concerned by this use case.

In case of composite identifier, the `@EmbeddedId` and `@Embeddable` annotations are used to declare the identifier and to map it as a property in the entity (see Listing 5.19). Figure 5.6 depicts the table concerned by this use case.

```
@Entity
@Table(name="customer")
public class Customer {
  @Embedded
  private Address address;
  ...
}

@Embeddable
public class Address{
  @Column
  private String city;

  @Column
  private String country
  ...
}
```

Listing 5.17: Example of embedded component declaration through JPA annotation.

```
@Entity
@Table(name="customer")
public class Customer {
  @Id
  @Column(name="id")
  private int customer_id;
  ...
}
```

Listing 5.18: Example of identifier declaration through JPA annotation.

```
@Entity
@Table(name="customer")
public class Customer {
  @EmbeddedId
  private CustomerId id;
  ...
}

@Embeddable
public class CustomerId implements Serializable {
  @Column(name="firstName")
  private String first_name;

  @Column(name="lastName")
  private String last_name;
  ...
}
```

Listing 5.19: Example of composite identifier declaration through JPA annotation.

```java
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="PAYMENT_TYPE", discriminatorType=DiscriminatorType.STRING)
@Table(name="PAYMENT")
public class Payment { ... }

@Entity
@DiscriminatorValue("CREDIT")
public class CreditCardPayment extends Payment { ... }

@Entity
@DiscriminatorValue("CASH")
public class CashPayment extends Payment { ... }

@Entity
@DiscriminatorValue("CHEQUE")
public class ChequePayment extends Payment { ... }
```

Listing 5.20: Example of entity inheritance declaration through JPA annotation.

Entity inheritance. JPA also supports entity inheritance. Inheritance is declared at the class level of the top level entity in the hierarchy by using the `@Inheritance` annotation. Moreover, a discriminator column and its discriminator values can be declared by using the `@DiscriminatorColumn` and `@DiscriminatorValue` annotations. Listing 5.20 depicts an example of inheritance declaration. Figure 5.7 depicts the table concerned by this use case.

The Hibernate schema extraction and the JPA schema extraction result in the extraction of, respectively, the Hibernate and JPA schemas. Since both are very similar, we merge them to obtain a unique condensed **ORM schema**. The latter contains a set of tables and columns concerned by an ORM mapping. All those tables/columns are annotated with meta-information about their mapping with the source code. For each table, we know the mapped persistent class and its path in the repository. For each column, we know the mapped class attribute and the class path in the repository. In other words, the ORM schema allows us to know which tables/columns are concerned by an ORM mapping, and for each mapped object, the accurate source code location where the mapping is declared.

5.2.2 Historization

At this step, the *extraction* phase is applied to each successive system version - let us denote n, the number of versions. In summary, for each system version, we extracted the corresponding database schema, the set of database accesses and the ORM schema.

The historization phase, depicted in Figure 5.8, consists of an *incremental* historization of the four inputs, namely (1) the n successive source code versions, (2) the n successive database schema versions, (3) the n successive ORM schema versions and (4) the n successive extracted sets of database accesses. This phase aims at computing a historical dataset that gathers information pertaining to the source code history, the database schema history, the ORM schema history and the database access history.

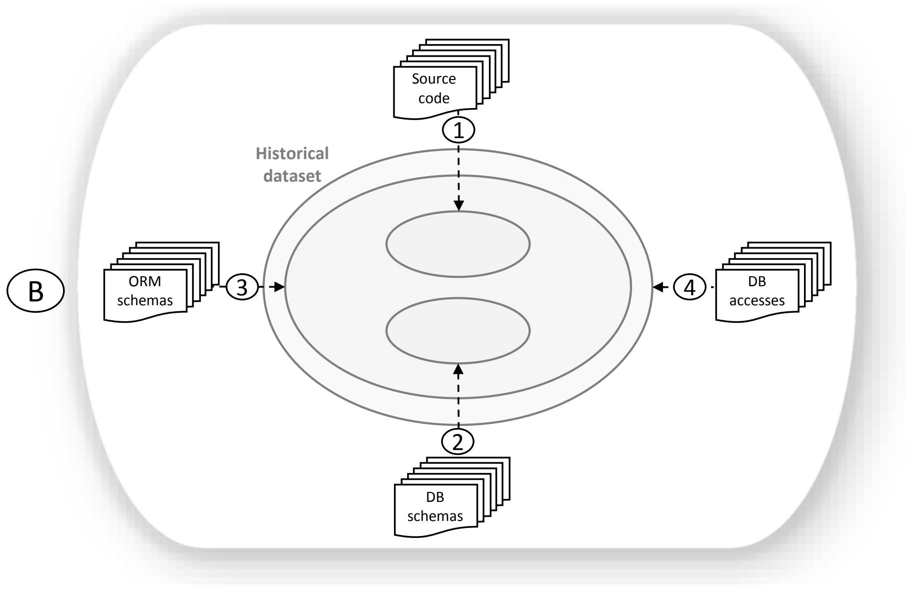


Figure 5.8: Phase (B) of our approach depicted in Figure 5.1: the Historization.

Source Code Historization

The source code historization focuses on the evolution of the source code over time. The aim of this process is to obtain historical information pertaining to the source code components. In particular, we focus on four types of source code components. The source code repository contains a set of Java **files**; each of them are identified by their file path in the repository and may contain one or several **class** declaration(s). Developers may declare a class as an *interface*. They can also define nested classes (a class within another class) characterized by a nested class path (e.g., `class1.class2.class3`). Each class may declare (1) **methods** characterized by their signature, and (2) **attributes** characterized by their name.

The source code historization computes historical data structured according to the Entity-Relationship model depicted in Figure 5.9. Each source code component is present in one or several system version(s); a system **version**, characterized by a commit date and a committer/developer id, is the central element which assigns a unique version identifier to the source code components, in order to facilitate tracing their evolution over time. At a given version, each source code component has its own definition at a particular position in the code, expressed as a couple of coordinates: a *begin line* and *column*, and an *end line* and *column*.

Database Schema Historization

The database schema historization focuses on the evolution of the database schema over time. The aim of this process is to obtain historical information pertaining to

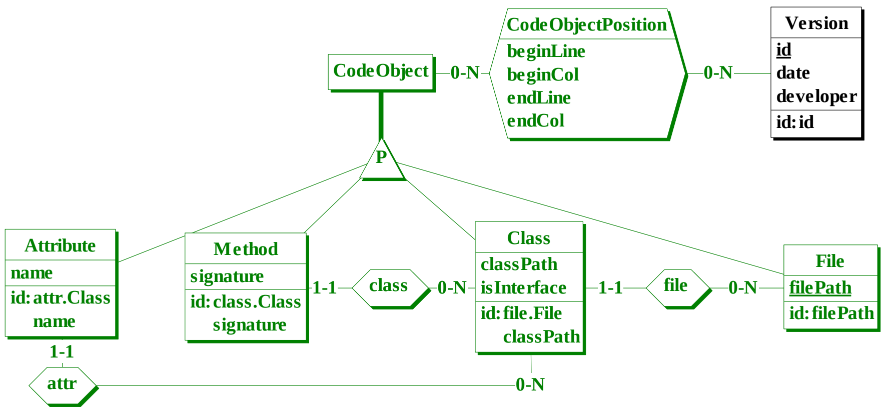


Figure 5.9: Entity-Relationship model of the source code historization.

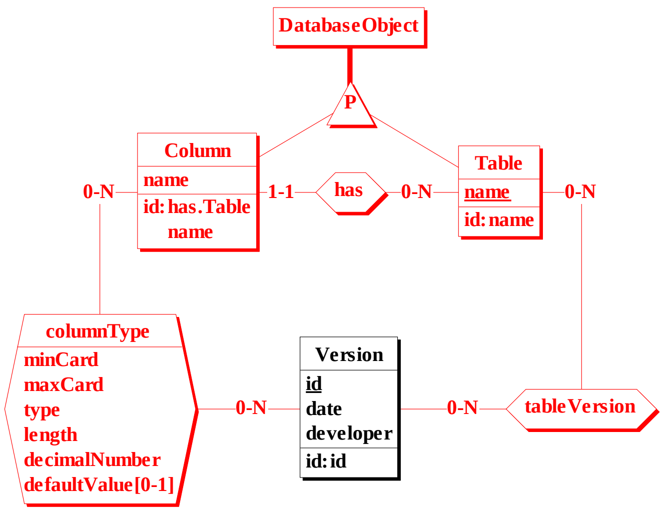


Figure 5.10: Entity-Relationship model of the database schema historization.

the database tables and columns. A database schema contains tables, identified by their name. Each table owns several columns characterized by their name.

The database schema historization computes historical data structured according to the Entity-Relationship model depicted in Figure 5.10. Each table/column is present in one or several system versions. A column is characterized by its type (its minimal and maximal *cardinality*, *data type*, *length*, *decimal number* and *default value*). The type of a particular column can be modified between versions.

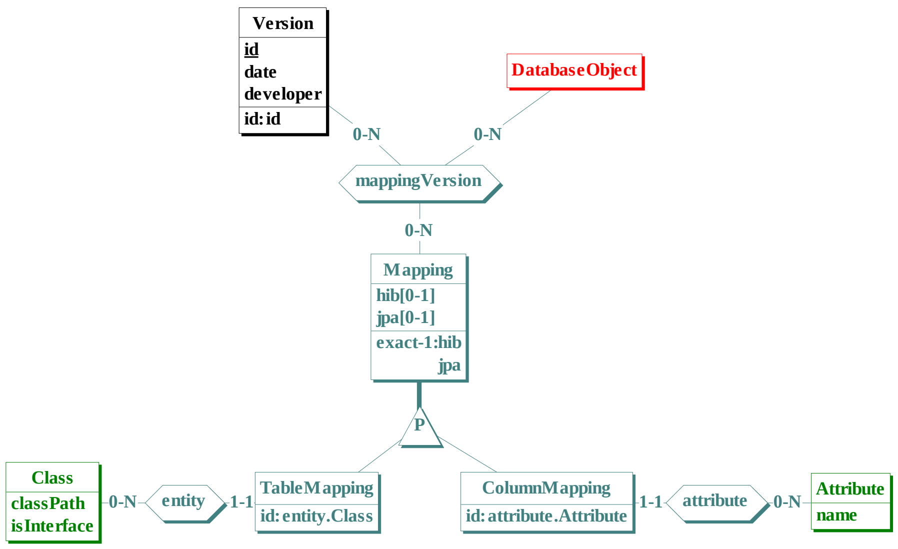


Figure 5.11: Entity-Relationship model of the ORM schema historization.

ORM Schema Historization

The ORM schema historization focuses on the evolution of the ORM usage over time. The aim of this process is to obtain historical information pertaining to the Hibernate and JPA mappings defined within the source code. In particular, we focus on **table** and **column** mappings. Each Hibernate/JPA mapping concerns a particular database table (respectively column) and a class (respectively attribute).

The ORM schema historization computes historical data structured according to the Entity-Relationship model depicted in Figure 5.11. Each ORM mapping is present in one or several system version(s).

Database Access Historization

The database access historization focuses on the evolution of the database usage in the source code over time. The aim of this process is to obtain historical information pertaining to the **database accesses**. In particular, we focus on *JDBC, Hibernate* and *JPA* accesses. A database access is present in one or several system versions. A database access is characterized by its minimal **program path** (i.e., calls to particular methods at a particular position in the code) necessary for creating and executing the given access. In addition, the program path and its position in the code may change between versions.

In particular, two kinds of database accesses are targeted, i.e., **queries** and **CRUD operations** (see Section 1.5). A query is characterized by its *original query* value; in case of HQL/JPQL query, the *translated SQL form* is another characteristic.

A query exists in one or several system versions and for each of those versions, it accesses a particular set of tables and columns, either *explicitly* or *implicitly* (e.g.,

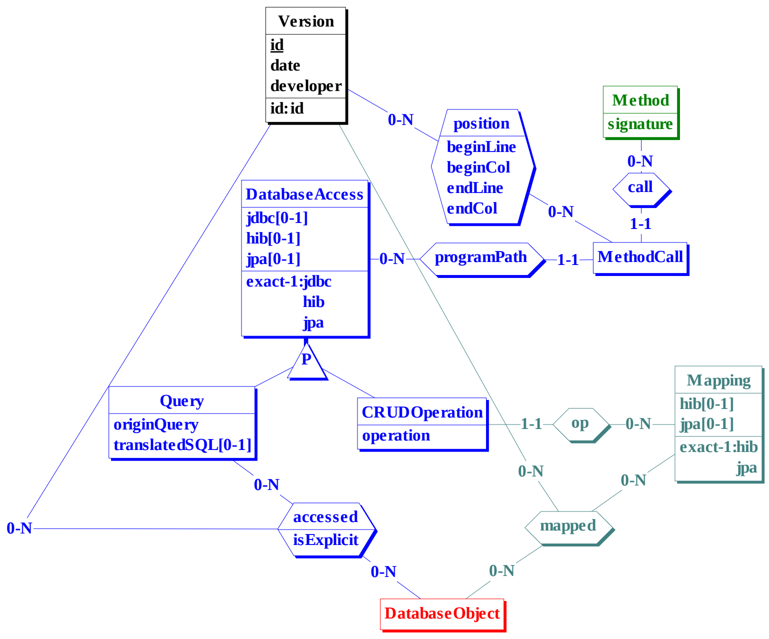


Figure 5.12: Entity-Relationship model of the database access historization.

provider_name is only explicitly accessed column of Provider in select * from Provider order by provider_name). A CRUD operation makes use of an ORM mapping to access a database table.

The database access historization computes historical data structured according to the Entity-Relationship model depicted in Figure 5.12.

Figure 5.13 shows the entire Entity-Relationship model obtained after historizing the 4 system artefacts. Querying this model permits us to analyze the accurate system state at any given version and to understand how the different artefacts have evolved and co-evolved over time. In summary, this data model allows us to answer **what**/**where**/**when**/**who**/**how** questions such as:

- Which tables/columns are accessed by a specific technology and where in the code?
- Which tables/columns are mapped by a specific ORM technology and where in the code?
- When was a specific technology introduced in the project?
- Which was the impact of the introduction of a new technology in the project?
- How do several database access technologies co-exist?
- Who is the most expert developer to achieve a particular evolution phase?
- Where are the error-prone database accesses which access removed/non-existing database objects?
- How do source code and database schema co-evolve over time?
- ...

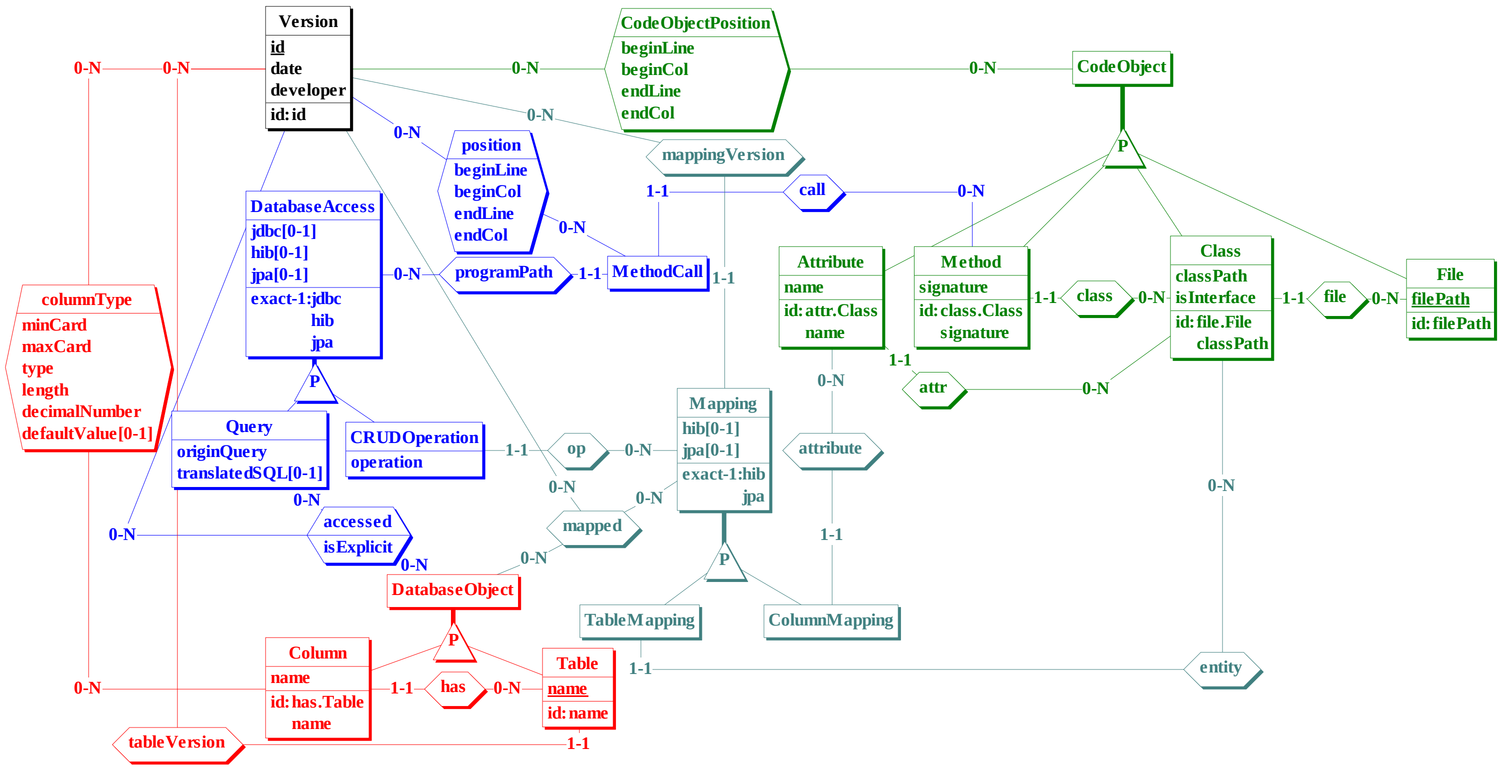


Figure 5.13: Entity-Relationship model of the historization of the four DISS artefacts.

5.3 Tool Support

We implemented this approach and developed a tool which permits to automatically compute the historical dataset, according to the ER model.

5.3.1 Extraction

To achieve this process, we implemented a set of analyzers in charge of executing the extraction phase on a particular DISS artefact. The extraction phase is then executed on each system version.

Database Schema Extraction

To implement the database schema extraction process, we reused the SQL code extraction, physical extraction and physical schema enrichment processes, described respectively in Sections 3.3.1, 3.3.2 and 3.3.3. As output, we obtain an enriched version of the database schema.

Database Access Extraction

To implement the database access extraction process, we reused our static analyzer allowing us to detect and recover JDBC, Hibernate and JPA accesses, as well as the code location where each access is created and executed. The static analyzer is detailed in Section 4.2.

Hibernate Schema Extraction

To achieve this task, we implemented an elaborated XML parser. The latter aims to visit the Hibernate mapping document in order to find the Hibernate mapping declarations. For each declared mapping, our parser automatically detects which table/-column is mapped to which class/attribute. As output, we obtain the corresponding Hibernate schema (i.e., the set of tables/columns concerned by a Hibernate mapping as well as the code location where the mapping is declared).

JPA Schema Extraction

In contrast to the Hibernate mappings, the JPA mappings are defined through annotations directly declared in the source code itself. To realize this task, we implemented a Java code analyzer; it constructs an abstract syntax tree and uses a visitor to navigate through the different Java nodes and expressions, looking for `annotation` nodes. Once detected, our analyzer determines which database objects (table/column) and code elements (class/attribute) are concerned by the mapping declaration. As output, we obtain the corresponding JPA schema.

Finally, the Hibernate and JPA schemas are merged as a condensed schema.

5.3.2 Historization

To realize this task, we made the choice to use a relational database to store the historical dataset. Relational DBMSs offer mechanisms to easily query this historical information, which motivates this choice. The historization of every DISS artefact (i.e., source code, database schema, ORM mappings and database accesses) consists in filling (a particular part of) this relational database. The `Version` table is the central table and brings a temporal dimension. This table contains information about each system version, i.e., the commit date and the developer's identifier who committed the version. Algorithm 4 fills the table with each system version and its characteristics.

```
procedure populateVersion(versions)
    for v ∈ versions do
        INSERT INTO Version VALUES (v.id, v.date, v.developer);
    end
```

Algorithm 4: Algorithm filling the `Version` table.

Source Code Historization

Figure 5.14 depicts the database schema fragment concerned by the source code historization. The `CodeObject` table materializes a particular source code object, i.e., a Java `File`, `Class`, `Method` or `Attribute`. The position of each code object in the parent file is recorded in the `CodeObjectPosition` table. This position is expressed as a couple of coordinates (the begin column and line, and the end column and line). This position can vary from a system version to another one; the couple of columns (`object_id`, `version_id`) identifies the code object and the system version in question.

Each code object has a unique `id`. However, they own a secondary identifier: a file is identified by its `filePath` in the repository; a class is identified by its owner file and `classPath`; a method is identified by its owner class and its `signature`; an attribute is identified by its owner class and its `name`.

Algorithm 5 describes the process filling the database with historical information about source code objects. It takes the list of system versions as input. The algorithm iterates over every system version. For each Java file of the current version, we check if the file was already encountered in the preceding versions and therefore already exists in the database (line 6). If it does not exist yet (lines 7-10), we insert it into the `CodeObject` and `File` tables. Otherwise, we retrieve the primary identifier of the existing file. Line 15 inserts the file position in the `CodeObjectPosition` table. The same process is applied to each encountered class/method/attribute.

Database Schema Historization

Figure 5.15 depicts the database schema fragment concerned by the database schema historization. The `DatabaseObject` table materializes a particular database

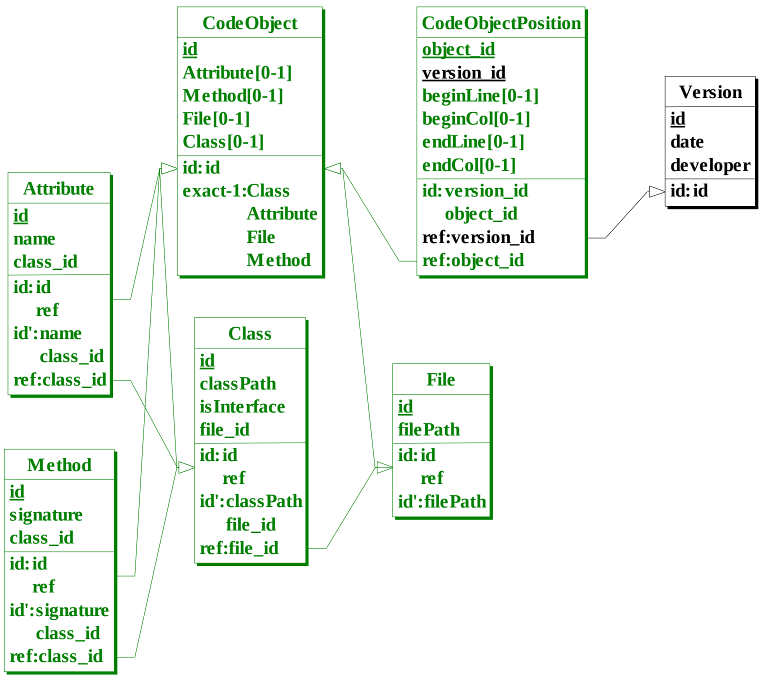


Figure 5.14: Logical schema of the source code historization.

schema object, i.e., a `Table` or `Column`. Each table/column has a unique identifier. However, they also have a secondary identifier: a table is identified by its `name`, and a column is identified by its parent table and its `name`. The versions of presence of each table are stored in the `tableVersions` table. The versions of presence of each column are stored in the `ColumnType` table; moreover, this table also contains information about the column type, which may vary from a version to another one.

Algorithm 6 describes the process filling the database with historical information about the tables and columns. It takes the list of system versions as input. The algorithm iterates over every system version. For each table of the current version, we check if the table already exists in the database (line 6). If it does not exist yet, we insert it into the `DatabaseObject` and `Tables` tables. Otherwise, we retrieve the primary identifier of the existing table. Line 15 adds the current version to its list of presence. Line 16 iterates over each column of the current table. Line 18 verifies if the column already exists in the database. If not yet, we insert it into the `DatabaseObject` and `Columns` tables. Line 27 stores the current type of the column in the `ColumnType` table.

ORM Schema Historization

Figure 5.16 depicts the database schema fragment concerned by the ORM schema historization. The `Mapping` table materializes a particular ORM mapping, i.e., a

```
procedure populateCodeObjects(versions)
    id = 0;
    for v ∈ versions do
        for file ∈ v.files do
            file2 = SELECT * FROM File where filePath:=file.filePath;
            if file2 = NULL then
                file.id = id;
                INSERT INTO CodeObject VALUES (file.id,NULL,NULL,1,NULL);
                INSERT INTO File VALUES (file.id, file.filePath);
                id = id + 1;
            end
            else
                file.id = file2.id;
            end
            INSERT INTO CodeObjectPosition VALUES (file.id,v.id,file.beginLine,file.beginCol,file.endLine,file.endCol);
            for class ∈ file.classes do
                class2 = SELECT * FROM Class WHERE classPath:=class.classPath AND file_id:=file.id;
                if class2 = NULL then
                    class.id = id;
                    INSERT INTO CodeObject VALUES (class.id,NULL,NULL,NULL,1);
                    INSERT INTO Class VALUES (class.id,class.classPath, class.isInterface,file.id);
                    id = id + 1;
                end
                else
                    class.id = class2.id;
                end
                INSERT INTO CodeObjectPosition VALUES
                   (class.id,v.id,class.beginLine,class.beginCol,class.endLine,class.endCol);
                for m ∈ class.methods do
                    m2 = SELECT * FROM Method WHERE signature:=m.signature AND class_id:=class.id;
                    if m2 = NULL then
                        m.id = id;
                        INSERT INTO CodeObject VALUES (m.id,NULL,1,NULL,NULL);
                        INSERT INTO Method VALUES (m.id,m.signature,class.id);
                        id = id + 1;
                    end
                    else
                        m.id = m2.id;
                    end
                    INSERT INTO CodeObjectPosition VALUES
                       (m.id,v.id,m.beginLine,m.beginCol,m.endLine,m.endCol);
                end
                for a ∈ class.attributes do
                    a2 = SELECT * FROM Attribute WHERE name:=a.name AND class_id:=class.id;
                    if a2 = NULL then
                        a.id = id;
                        INSERT INTO CodeObject VALUES (a.id,1,NULL,NULL,NULL);
                        INSERT INTO Attribute VALUES (a.id,a.name,class.id);
                        id = id + 1;
                    end
                    else
                        a.id = a2.id;
                    end
                    INSERT INTO CodeObjectPosition VALUES
                       (a.id,v.id,a.beginLine,a.beginCol,a.endLine,a.endCol);
                end
            end
        end
    end
```

Algorithm 5: Algorithm populating the database with historical information about source code evolution.

`TableMapping` or a `ColumnMapping`. Each mapping has a unique identifier. In addition, they have a secondary identifier: a table mapping is identified by the class where it is declared and the used `technology` (Hibernate/JPA); a column mapping is identified by the attribute on which the mapping is declared and the `technology`. A mapping can exist in several system versions; the `MappingVersion` table contains the list of presence of every mapping. A mapping is declared between a code object (class or attribute) and a set of database objects (tables or columns). However, the set of mapped objects can change over time; therefore, the `MappingAccess` table

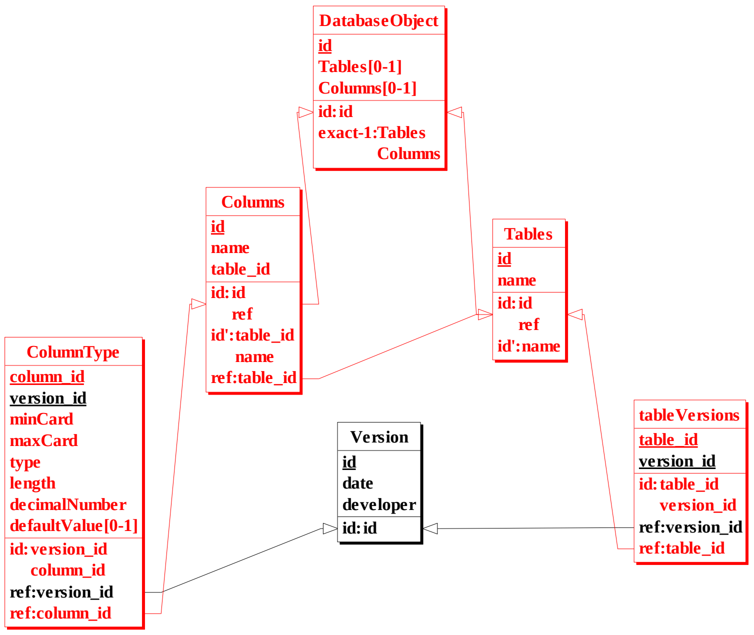


Figure 5.15: Logical schema of the database schema historization.

```
procedure populateDBSchema(versions)
    id = 0;
    for v ∈ versions do
        for table ∈ v.tables do
            table2 = SELECT * FROM Tables WHERE name:=table.name;
            if table2 = NULL then
                table.id = id;
                INSERT INTO DatabaseObject VALUES (table.id,1,NULL);
                INSERT INTO Tables VALUES (table.id,table.name);
                id = id + 1;
            end
            else
                table.id = table2.id;
            end
            INSERT INTO tableVersions VALUES (table.id,v.id);
            for col ∈ table.columns do
                col2 = SELECT * FROM Columns where name:=col.name AND table_id:=table.id;
                if col2 = NULL then
                    col.id = id;
                    INSERT INTO DatabaseObject VALUES (col.id,NULL,1);
                    INSERT INTO Columns VALUES (col.id,col.name,table.id);
                    id = id + 1;
                end
                else
                    col.id = col2.id;
                end
                INSERT INTO ColumnType VALUES
                  (col.id,v.id,col.minCard,col.maxCard,col.type,col.length,col.decimal,col.defaultValue);
            end
        end
    end
```

Algorithm 6: Algorithm populating the database with historical information about database schema evolution.

contains, for each version of presence, the set of mapped tables/columns.

Sometimes, some *special* mappings can be declared; let us imagine cases where

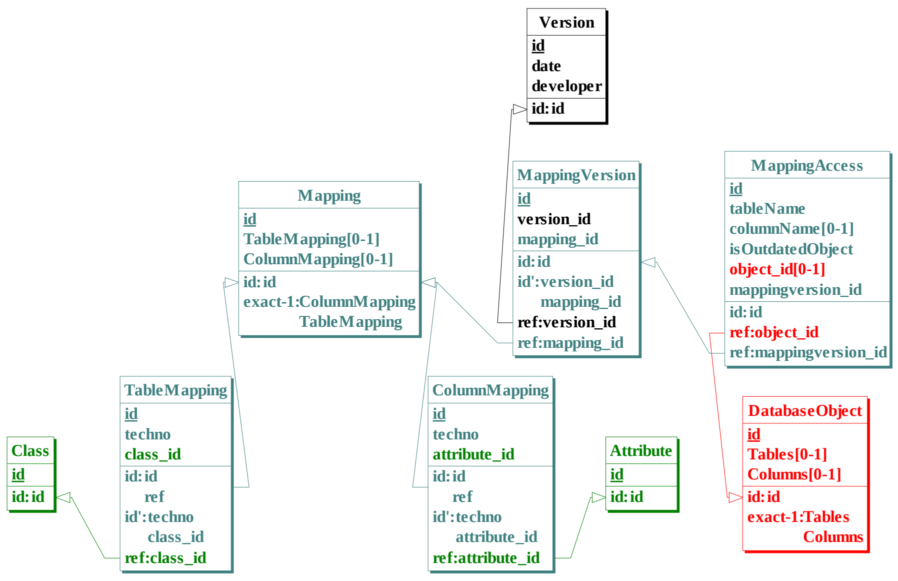


Figure 5.16: Logical schema of the ORM schema historization.

developers define an *incorrect* mapping between a class/attribute and a misspelled or deleted table/column. Storing such information can be precious for detecting error-prone database-related code locations. This is why we added some columns to the `MappingAccess` table; the `tableName` and `columnName` columns store the name of the mapped table/column. If the mapped object is an actual table/column the `object_id` column will contain the actual identifier of the mapped table/column. Otherwise, the value will be null. In the case where the mapping is declared with a removed (or later created) table/column, the `isOutdatedObject` column will be equals to *true*. Otherwise, *false*.

Algorithm 7 describes the process filling the database with historical information about ORM mappings. It takes the list of system versions as input. The algorithm iterates over every system version. For each table mapping of the current version, we check if the mapping already exists in the database (line 10). If not, we insert it into the `Mapping` and `TableMapping` tables. Otherwise, we retrieve the primary identifier of the existing mapping. Line 19 adds the current version to its list of presence. For each table concerned by the mapping, line 25 checks if the table is an actual table registered in the database. If yes, line 27 checks if the table is present in the current database schema version. Line 29 inserts information about each mapped table.

From line 33, the algorithm iterates over every column mapping of the current version. Line 38 verifies the existence of the column mapping in the database. Line 47 adds the current version to its list of presence. For each column concerned by the mapping, line 57 checks if the column is an actual column registered in the database. If it is, line 59 checks if the column is present in the current database schema version.

Line 61 inserts information about each mapped column.

```
1  procedure populateORMSchema(versions)
2      id = 0;
3      mv_id = 0;
4      ma_id = 0;
5      for v ∈ versions do
6          for tm ∈ v.tableMappings do
7              file = SELECT * FROM File WHERE filePath:=tm.filePath;
8              class = SELECT * FROM Class where classPath:=tm.classPath AND file_id:=file.id;
9              tm2 = SELECT * FROM TableMapping techno:=tm.techno AND class_id:=class.id;
10             if tm2 = NULL then
11                 tm.id = id;
12                 INSERT INTO Mapping VALUES (tm.id,1,NULL);
13                 INSERT INTO TableMapping VALUES (tm.id,tm.techno,class.id);
14                 id = id + 1;
15             end
16             else
17                 tm.id = tm2.id;
18             end
19             INSERT INTO MappingVersion VALUES (mv_id,v.id,tm.id);
20             mv_id = mv_id + 1;
21             for tMap ∈ tm.tables do
22                 table = SELECT * FROM Tables where name:=tMap.name;
23                 object_id = NULL;
24                 isOutdated = false;
25                 if table ≠ NULL then
26                     object_id = table.id;
27                     isOutdated = ((SELECT 1 FROM TableVersions where table_id := object_id AND version_id := v.id)
                           = NULL);
28                 end
29                 INSERT INTO MappingAccess VALUES (ma_id,tm.table.name,NULL,isOutdated,object_id,mv_id);
30                 ma_id = ma_id + 1;
31             end
32         end
33         for cm ∈ v.columnMappings do
34             file = SELECT * FROM File where filePath:=cm.filePath;
35             class = SELECT * FROM Class where classPath:=cm.classPath AND file_id:=file.id;
36             attribute = SELECT * FROM Attribute WHERE name:=cm.attributeName AND class_id:=class.id;
37             cm2 = SELECT * FROM ColumnMapping WHERE techno:=cm.techno AND attribute_id:=attribute.id;
38             if cm2 = NULL then
39                 cm.id = id;
40                 INSERT INTO Mapping VALUES (cm.id,NULL,1);
41                 INSERT INTO ColumnMapping VALUES (cm.id,cm.techno,attribute.id);
42                 id = id + 1;
43             end
44             else
45                 cm.id = cm2.id;
46             end
47             INSERT INTO MappingVersion VALUES (mv_id,v.id,cm.id);
48             mv_id = mv_id + 1;
49             for cMap ∈ cm.columns do
50                 table = SELECT * FROM Tables WHERE name:=cMap.table.name;
51                 column = NULL;
52                 if table ≠ NULL then
53                     column = SELECT * FROM Columns WHERE name:=cMap.name AND table_id:=table.id;
54                 end
55                 object_id = NULL;
56                 isOutdated = false;
57                 if column ≠ NULL then
58                     object_id = column.id;
59                     isOutdated = ((SELECT 1 FROM ColumnType where column_id := object_id AND version_id :=
                           v.id) = NULL);
60                 end
61                 INSERT INTO MappingAccess VALUES
                     (ma_id,tm.table.name,tm.column.name,isOutdated,object_id,mv_id);
62                 ma_id = ma_id + 1;
63             end
64         end
65     end
```

Algorithm 7: Algorithm populating the database with historical information about ORM schema evolution.

Database Access Historization

Figure 5.17 depicts the database schema fragment concerned by the database access historization. The `DatabaseAccess` table materializes a particular database access, i.e., a `Query` or `CRUDOperation`. Each database access is characterized by its program path; namely a database access appears in the program code as `MethodCall` to particular methods (at a particular position in the code). This position can change from a version to another one; the `MethodCallPosition` table records its position at a given version.

Each access has a unique identifier. In addition, we decided to affect them an *implicit* secondary identifier; a query is identified by its query value (`originQuery`), the access `technology` (JDBC/Hibernate/JPA) and its program path. A CRUD operation is identified by the used mapping, the access `operation` (insertion/selection/update/deletion) and its program path.

A JPQL/HQL query has a corresponding translated SQL form . This translated SQL form can change according to the system version. The `VersionQuery` table stores the `translatedSQL` value at a given version. However, the SQL translation may sometimes fail because of a syntax error; in such a case, the `translatedSQL` column will be null. Since the translated SQL value may change from a system version to another one, the set of tables/columns accessed by the query may change too. The `QueryAccess` table contains the set of objects accessed at a given version. In addition, the `isExplicit` column indicates if the object is explicitly or implicitly accessed. The `isOutdatedObject` column indicates if the query attempts to access a removed/non-existing table/column. This field is valuable to detect error-prone database-related code locations.

A CRUD operation can also exist in several system versions. The `VersionCRUD` table contains the list of presence of each mapping. The `isOutdatedMapping` column indicates if the concerned mapping is up-to-date or not.

Algorithm 8 describes the process filling the database with historical information about the database accesses. It takes the list of system versions as input. The algorithm iterates over every system version. For each query, line 5 checks if it already exists in the database (the details of the `loadQuery` procedure are given in Algorithm 9). If it does not, we insert it into the `DatabaseAccess` and `Query` tables. Lines 10-17 register the program path in the `MethodCall` table. Since the program path position of an access can change from a version to another one, line 24 records its position in the `MethodCallPosition` table. Line 26 adds the current version to the list of presence of the query. Lines 30 and 36 register respectively, the tables and columns accessed by the query, with the verification that the object is up-to-date.

Line 40 iterates over each CRUD operation. Line 41 checks if the CRUD operation already exists in the database (the details of the `loadCRUD` procedure are given in Algorithm 10). If it does not, we insert it into the `DatabaseAccess` and `CRUDOperation` tables. Line 55 registers the access program path, while line 65 registers its position in the current code version. Line 68 adds the current version to its list of presence, with the verification that the concerned mapping is up-to-date.

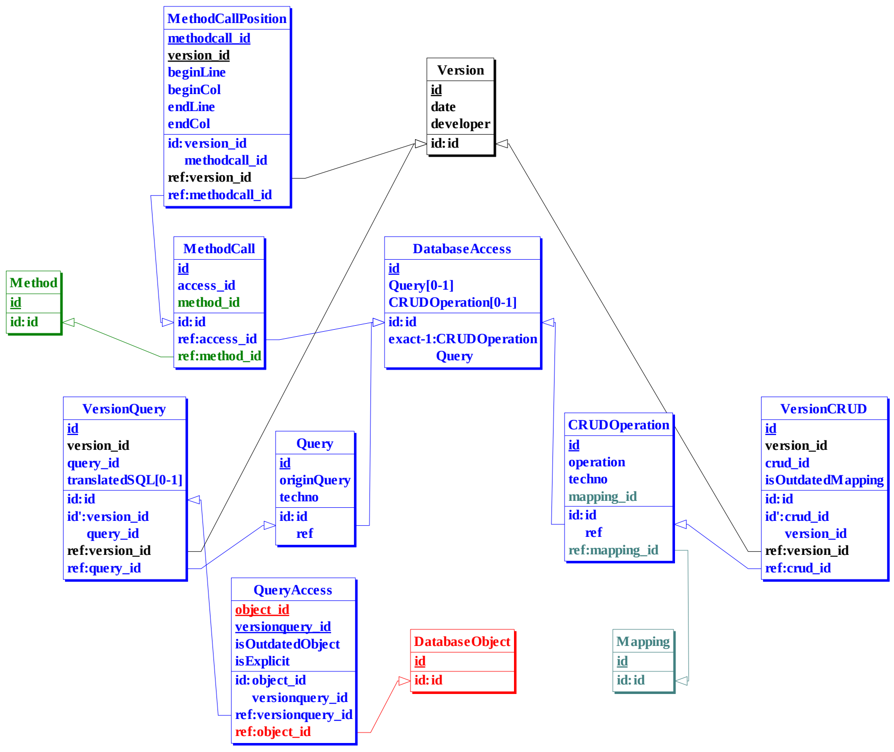


Figure 5.17: Logical schema of the database access historization.

```
procedure populateDBAccesses(versions)
    id = 0,mc_id = 0,vq_id = 0,vc_id = 0;
    for v ∈ versions do
        for query ∈ v.queries do
            query2 = loadQuery(query);
            if query2 = NULL then
                query.id = id;
                INSERT INTO DatabaseAccess VALUES (query.id,1,NULL);
                INSERT INTO Query VALUES (query.id,query.originQuery,query.techno);
                for methodCall ∈ query.programPath do
                    methodCall.id = mc_id;
                    file = SELECT * FROM File where filePath:=methodCall.filePath;
                    class = SELECT * FROM Class WHERE classPath:=methodCall.classPath AND file_id:=file.id;
                    method = SELECT * FROM Method WHERE signature:=methodCall.signature AND
                      class_id:=class.id;
                    INSERT INTO MethodCall VALUES (mc_id,id,method.id);
                    mc_id = mc_id + 1;
                end
                id = id + 1;
            end
            else
                query.id = query2.id;
            end
            for mc ∈ query.programPath do
                INSERT INTO MethodCallPosition VALUES
                  (mc.id,v.id,mc.beginLine,mc.beginCol,mc.endLine,mc.endCol);
            end
            INSERT INTO VersionQuery VALUES (vq_id,v.id,query.id,query.translatedSQL);
            for table ∈ query.accessedTables do
                t = SELECT * FROM Tables WHERE name:=table.name;
                isOutdated = ((SELECT 1 FROM TableVersions WHERE table_id := t.id AND version_id := v.id) = NULL);
                INSERT INTO QueryAccess VALUES (t.id,vq_id,isOutdated,true);
            end
            for column ∈ query.accessedColumns do
                t = SELECT * FROM Tables WHERE name:=column.table.name;
                c = SELECT * FROM Columns WHERE name:=column.name AND table_id:=t.id;
                isOutdated = ((SELECT 1 FROM ColumnType WHERE column_id := c.id AND version_id := v.id) =
                  NULL);
                INSERT INTO QueryAccess VALUES (c.id,vq_id,isOutdated,c.isExplicit);
            end
            vq_id = vq_id + 1;
        end
        for crud ∈ v.crudOperations do
            crud2 = loadCRUD(crud);
            if crud2 = NULL then
                crud.id = id;
                INSERT INTO DatabaseAccess VALUES (crud.id,NULL,1);
                file = SELECT * FROM File WHERE filePath:=crud.filePath;
                class = SELECT * FROM Class WHERE classPath:=crud.classPath AND file_id:=file.id;
                mapping = SELECT * FROM TableMapping WHERE techno:=crud.techno AND class_id:=class.id;
                crud.mapping = mapping;
                INSERT INTO CRUDOperation VALUES (crud.id,crud.operation,crud.techno,mapping.id);
                for methodCall ∈ crud.programPath do
                    methodCall.id = mc_id;
                    file = SELECT * FROM File WHERE filePath:=methodCall.filePath;
                    class = SELECT * FROM Class WHERE classPath:=methodCall.classPath AND file_id:=file.id;
                    method = SELECT * FROM Method WHERE signature:=methodCall.signature AND
                      class_id:=class.id;
                    INSERT INTO MethodCall VALUES (mc_id,id,method.id);
                    mc_id = mc_id + 1;
                end
                id = id + 1;
            end
            else
                crud.id = crud2.id;
                crud.mapping = crud2.mapping;
            end
            for mc ∈ crud.programPath do
                INSERT INTO MethodCallPosition VALUES
                  (mc.id,v.id,mc.beginLine,mc.beginCol,mc.endLine,mc.endCol);
            end
            isOutdated = ((SELECT 1 FROM MappingVersion WHERE version_id := v.id AND mapping_id :=
              crud.mapping.id) = NULL);
            INSERT INTO VersionCRUD VALUES(vc_id,v.id,crud.id,isOutdated);
            vc_id = vc_id + 1;
        end
    end
```

Algorithm 8: Algorithm populating the database with historical information about the evolution of the database accesses.

```
procedure loadQuery(query)
    for query2 ∈ (SELECT * FROM Query WHERE originQuery:=query.originQuery
      AND techno:=query.techno) do
        programPath2 = SELECT * FROM MethodCall WHERE access_id:=query2.id;
        if query.programPath = programPath2 then
            return query2;
        end
    end
    return NULL;
```

Algorithm 9: Algorithm searching for the existence of a given query in our database, based on the equality of the original query value, the used technology and the program path.

```
procedure loadCRUD(crud)
    for crud2 ∈ (SELECT * FROM CRUDOperation WHERE
      mapping_id:=crud.mapping_id AND operation:=crud.operation) do
        programPath2 = SELECT * FROM MethodCall WHERE access_id:=crud2.id;
        if crud.programPath = programPath2 then
            return crud2;
        end
    end
    return NULL;
```

Algorithm 10: Algorithm searching for the existence of a given CRUD operation in our database, based on the equality of the used mapping, the operation and the program path.

5.4 Application to Real-Life Systems

In this Section, we show the benefits of our approach by studying the evolution of three large open-source Java systems, i.e., OSCAR, OpenMRS and Broadleaf. To perform this study, we apply the approach to each system in order to measure how some characteristics of database usage in these systems evolve over time. Our objective is to understand, at a fine-grained level, how the systems evolve over time, how the database and code co-evolve and how several technologies may co-exist into the same system.

5.4.1 Analysis Background

For all the systems, we applied our automatic historical analysis approach, presented in the previous section, on the source code of selected versions from the version control systems. Firstly, we picked the initial commits and then we went on through the next versions and selected those that were at least 15 days from the last selected version and contained at least 500 modified lines[3]. We have thus a snapshot of the

[3]Those values were arbitrarily chosen, based on our observations. We claim that 500 lines modified between successive considered versions constitute significant changes to study the evolution of those systems.

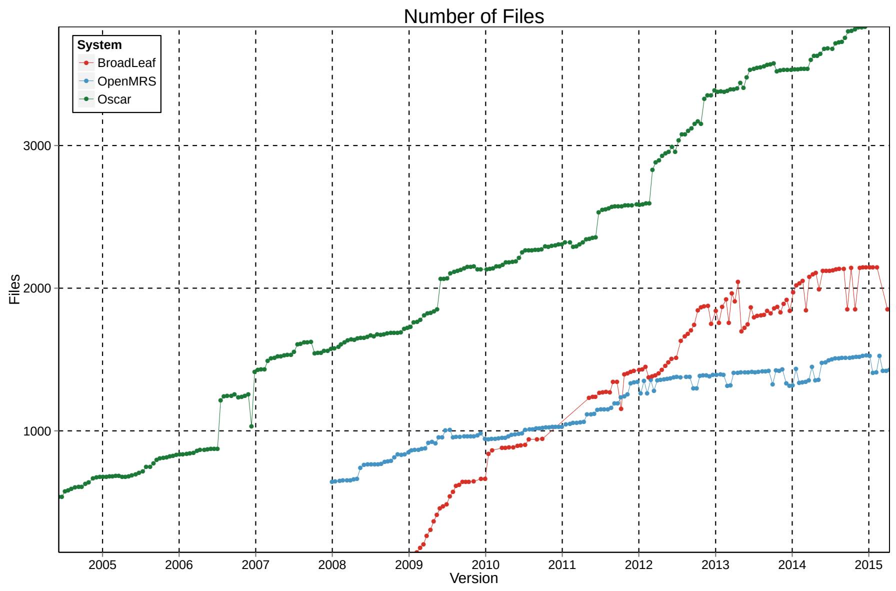


Figure 5.18: Evolution over time of the number of files in each system.

state of each system in every 2-3 weeks of its development. We respectively selected 242, 164 and 118 versions for OSCAR, OpenMRS and Broadleaf

Finally, we applied our *extraction* process to each version and *historized* them. As a result, we have a database for each of the three systems, filled with historical information (as implemented in Section 5.3.2).

5.4.2 Results

With the exploitation of our databases, we studied the three data-intensive systems. Through the measurement of database usage characteristics, we investigated and understood how the systems evolve over time, how the database and source code co-evolve and how several technologies co-exist within a same system. Through our study, we analyzed the history of each system and pointed out that each of them has a specific design and evolution.

As is customary for many open source projects, the number of code files of each system grows more or less linearly over time (see Figure 5.18).

Each of the three considered systems appears to have its own specific database schema growth trend. Figure 5.19 depicts, for each system, the evolution of the number of tables. While the schema of OSCAR continuously grows over time, OpenMRS and Broadleaf seem to have a more periodic growth. There are fewer changes in the OpenMRS schema. Its developers rarely remove tables and columns and have a well-prepared extension phase with the addition of 21 tables in November 2011. Except for this period of growth, the schema size remains constant. After an initial phase of growth (up to October 2009), the Broadleaf schema remains more or less

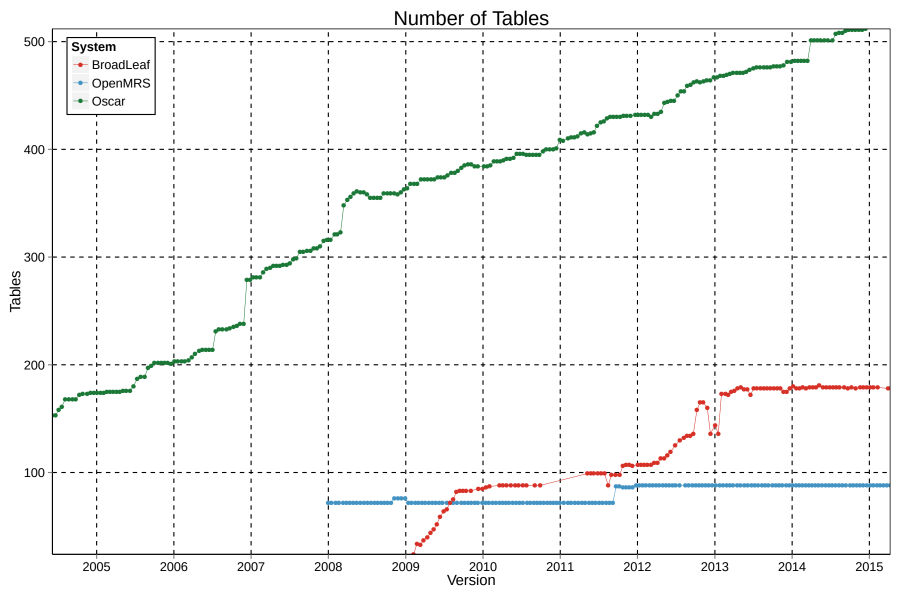


Figure 5.19: Evolution over time of the number of tables in each system.

stable until June 2012. From there until February 2013, the schema undergoes a strong growth, followed again by a stable phase. Nevertheless, during that stable period, we observe some schema changes with successive additions and removals of tables. A more detailed analysis revealed that those changes correspond to a renaming phase of some tables.

In order to study the co-evolution between the source code and the database schema, we focused on three artifacts: (1) the tables that are accessed and the way to access them (Figure 5.20); (2) the code locations and files accessing the database (respectively Figure 5.21 and Figure 5.22); and (3) the distribution of mappings across ORM technologies (Figure 5.23).

OSCAR

(1) Initially, OSCAR only used the JDBC API to access the database. In August 2006, Hibernate was introduced in the system but the number of JDBC-accessed tables did not decrease. In April 2008, JPA appears but remains infrequently used up to March 2012. While JDBC was the prevailing technology (in terms of accessed tables) until there, a massive migration phase happened and JPA became the main technology, with a decrease of Hibernate and JDBC usage. Today, the three technologies still co-exist and we observe that many tables in the database are accessed by at least two technologies, which may be considered as a sign of bad coding practices, or a technology migration process that is still ongoing.

(2) The database access location distribution follows a different trend. Until the

introduction of JPA, there was a majority of Hibernate access locations. Once JPA was introduced, the number of Hibernate and JDBC access locations progressively decreased. We also analyzed the distribution of database technologies across Java files. Here again, the distribution over time confirms a massive migration phase in March 2012 with the explosion of the number of files that access the database via JPA and the decrease of the number of files using JDBC or Hibernate. Some files allow accessing the database via both JDBC and Hibernate and might indicate bad coding practices or non-ended migration.

(3) The observed migration phase also impacts the ORM mappings defined between the Java classes and the database tables. The majority of the Hibernate mappings has been replaced by JPA mappings. Nevertheless, a big part of the database schema remains unmapped. A small set of tables contain both Hibernate and JPA mappings, which is a potential problem that should probably be fixed in the future.

OpenMRS

(1) Since the beginning, OpenMRS combined JDBC and Hibernate to query its database. However, while a majority of tables are accessed via Hibernate, only a few tables are accessed through JDBC. In November 2011, almost all the 21 added tables are exclusively accessed via Hibernate. Hibernate clearly appears as the main technology but it is interesting to point out that some tables are accessed via both JDBC and Hibernate during the whole system's life.

(2) The access location point distribution confirms that Hibernate is the main technology. The number of JDBC locations is much lower than the Hibernate locations and the number of Hibernate files is the predominant part. What is more surprising is the increasing number of JDBC files in comparison to the limited number of tables accessed via JDBC.

(3) Since we observed that Hibernate was the main technology and also the only used ORM, it is not astounding to see that the majority of tables are mapped to Java classes.

Broadleaf

(1) Broadleaf uses JPA for accessing its database from the programs source code. The number of non-accessed tables remains very high during the whole system's life. Moreover we observe a stabilization of that number from February 2013 (one can see the same trend regarding the size of the database schema).

(2) The access location point distribution also follows the same trend with that stabilization in February 2013. What is more interesting is that Broadleaf looks very well designed and divided from the start of the project with an average of one database-accessing file per table.

(3) The ORM mappings are defined on the majority of the database tables and do not evolve anymore since the stabilization period.

5.4.3 Discussion

OSCAR is a frequently changing system. Code and database schema have continuously evolved. It seems clear that the introduction of a new technology (Hibernate and later JPA) was aimed to replace the previous one; we can clearly identify the decrease in the usage of JDBC (resp. Hibernate) after the introduction of Hibernate (resp. JPA).We noticed that those migrations are still ongoing, as can be witnessed by the presence of tables accessed by several technologies, as well as by the presence of several technologies in a same file. A more blatant example is the co-existence of Hibernate and JPA mappings for some tables. Furthermore, the three technologies (JDBC, Hibernate and JPA) have co-existed for several years and make code and database evolution more complex and time-consuming. OSCAR developers even admit it: one "*can use a direct connection to the database via a class called DBHandler, use a legacy Hibernate model, or use a generic JPA model. As new and easier database access models became available, they were integrated into OSCAR. The result is that there is now a slightly noisy picture of how OSCAR interacts with data in MySQL.*" [Ruttan, 2008].

Compared to OSCAR, OpenMRS is a less prone to changes. Its code has increasingly evolved over time but there were very fewer changes in the database schema, which has remained quite stable over the years. These changes seem to be periodic and better anticipated. Most of those changes are applied at the same versions. Moreover, one can notice that database objects are rarely removed from the schema. Another major difference with OSCAR is that JDBC and Hibernate co-exist from the beginning of the project and are complementary: no technology aims to substitute the other.

Among the three systems, Broadleaf seems to be the one with the simplest design. Indeed, Broadleaf only uses JPA to communicate with the database. Moreover, Broadleaf looks well structured and easy to maintain. The detection of database locations in the code requires little effort since the lines of code that access tables are usually regrouped into a single file.

5.5 Concluding Remarks

In this Chapter, we presented a historical analysis approach collecting evolution history information of several DISS artefacts. While the *extraction* phase of our approach (described in Section 5.2.1) is specifically designed for Java systems, the *historization* phase relies on a data model detailed in Section 5.2.2 and could become totally technology-independent with some minor adaptations.

We then motivated the benefits of this approach on three real-life systems; we show how exploiting this historical information can significantly aid at analyzing, at a fine-grained level, how the systems evolved over time. During our analysis, we made interesting observations; we observed, among others, that the very same

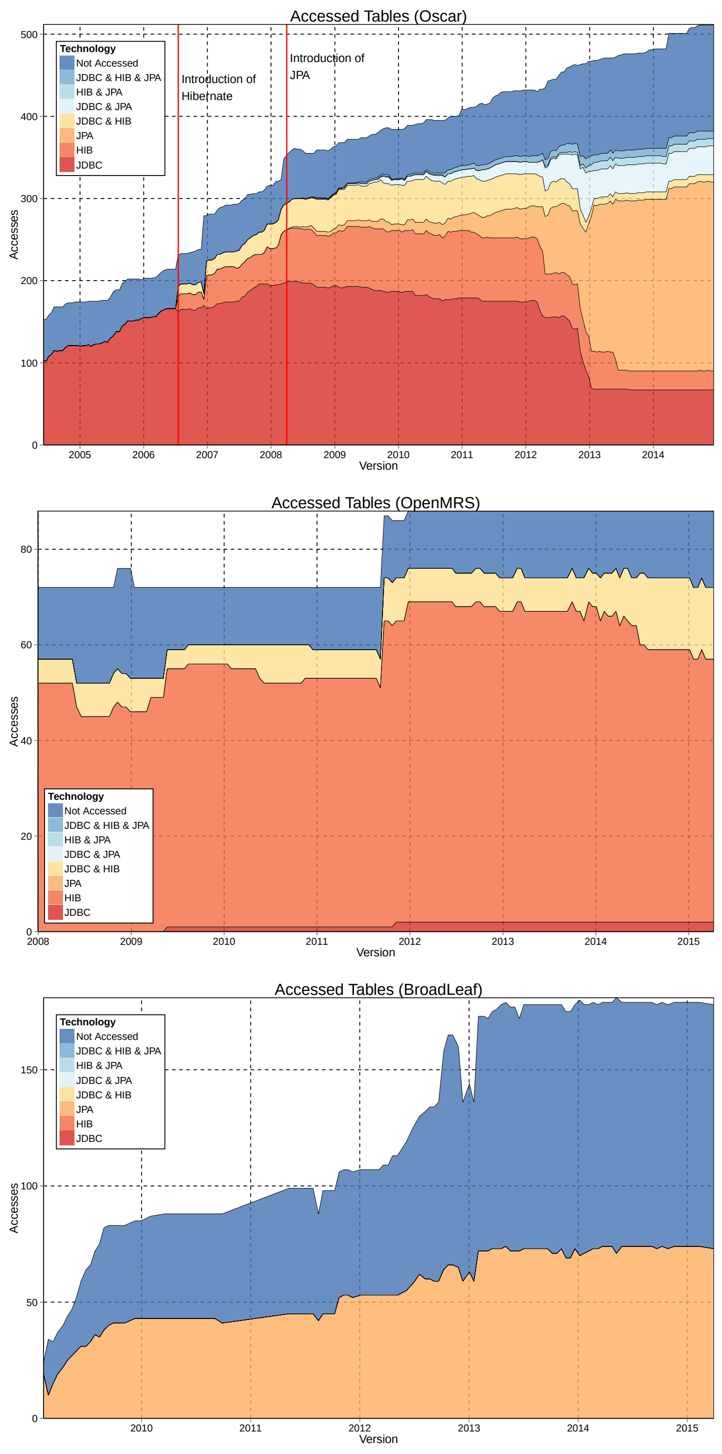


Figure 5.20: Distribution of the accessed tables across the technologies.

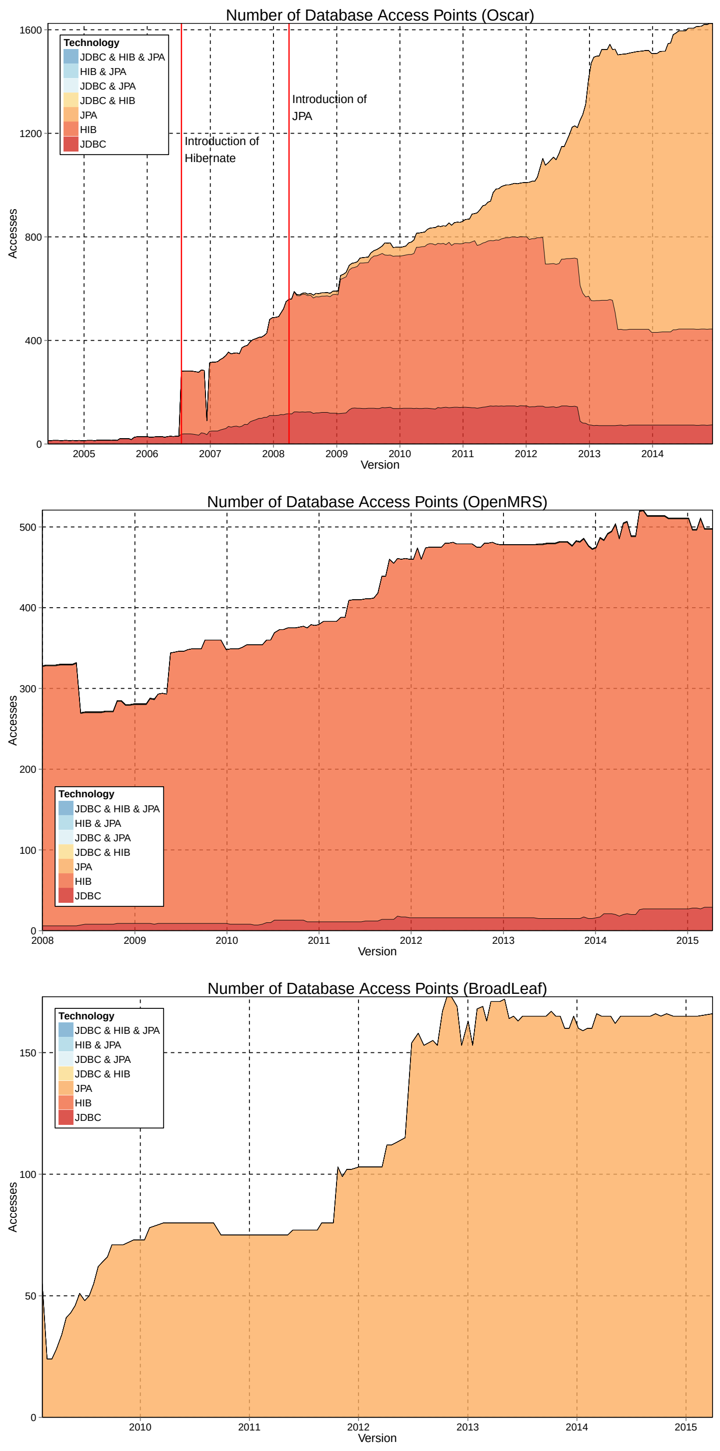


Figure 5.21: Database access point distribution.

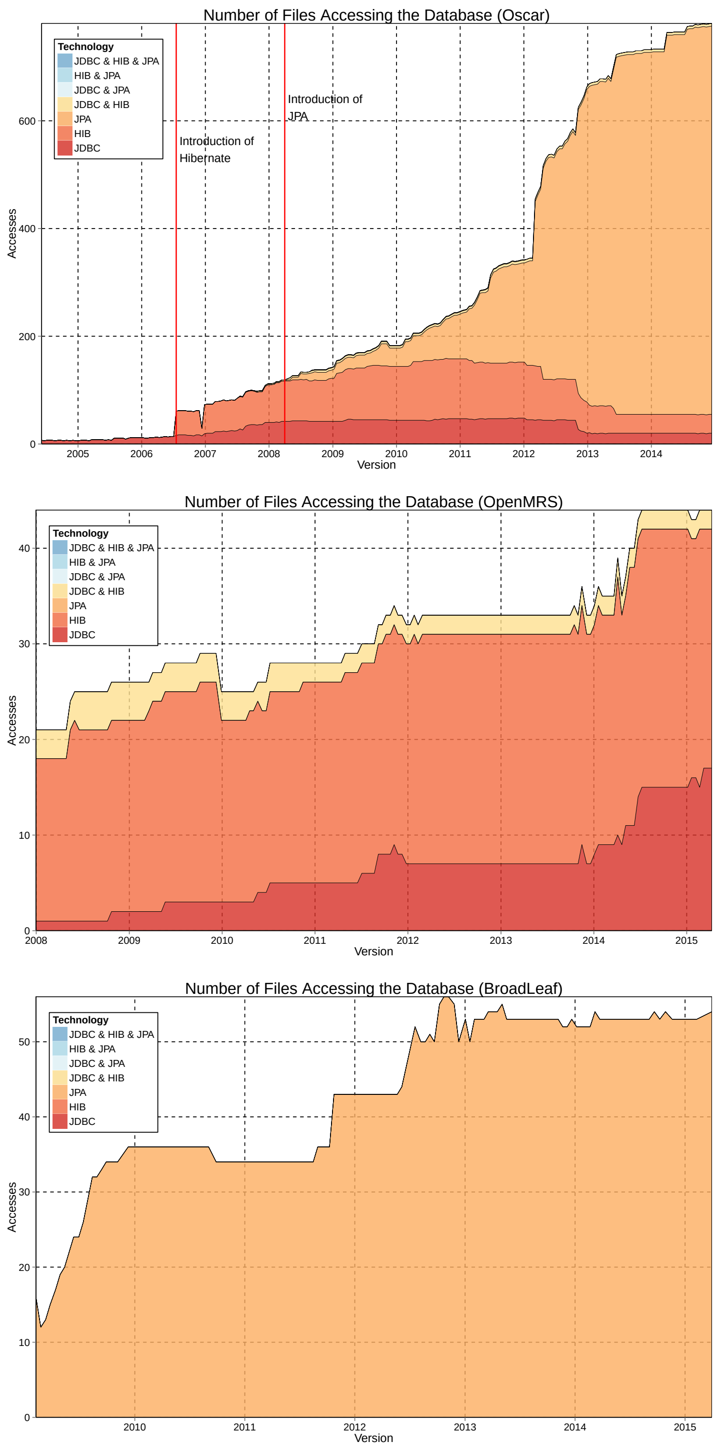


Figure 5.22: The distribution of the files accessing the database.

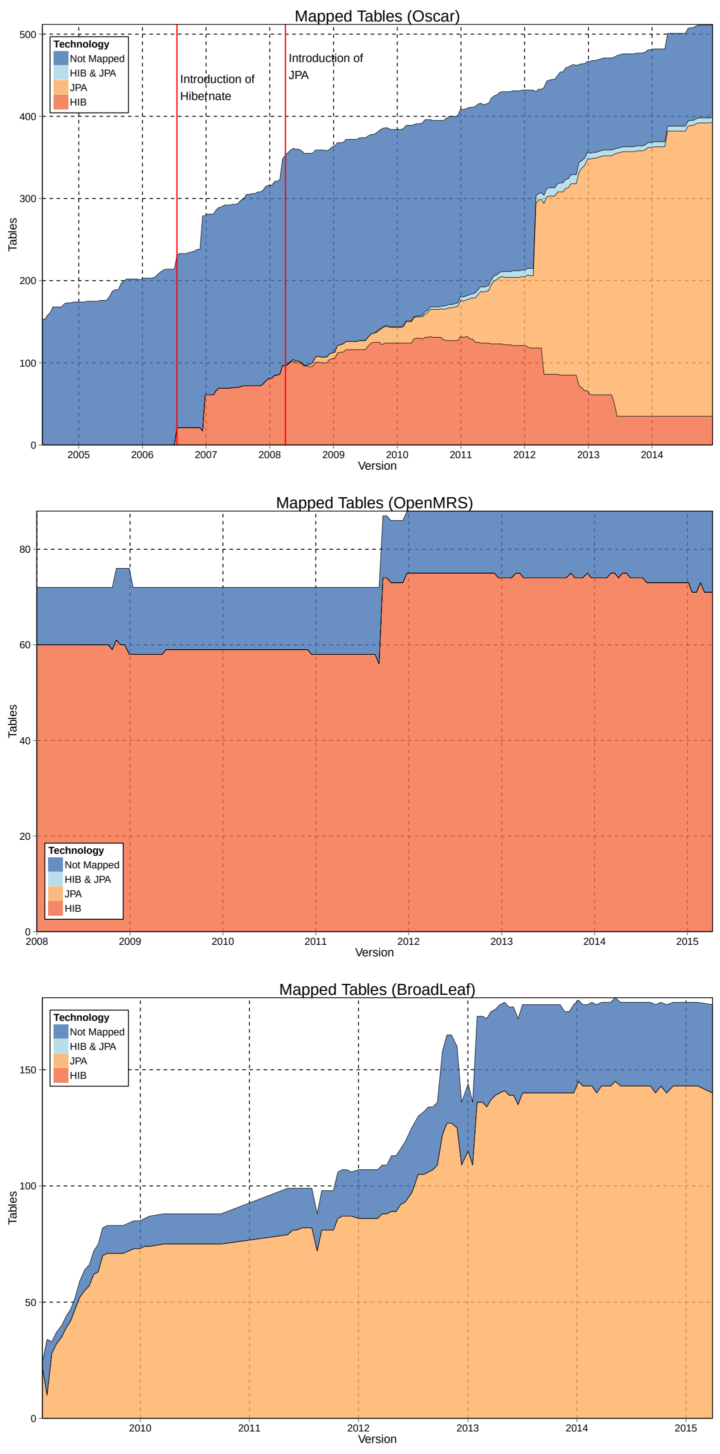


Figure 5.23: Distribution of the mapped tables across the technologies.

tables could be accessed by different data manipulation technologies within the programs. We also observed that database schemas may quickly grow over time, most schema changes consisting in adding new tables and columns. Finally, we saw that a significant subset of database tables and columns are not accessed (any longer) by the application programs. The presence of such "dead" schema elements might suggest that the co-evolution of schema and programs is not always a trivial process for the developers. The developers seem to refrain from evolving a table in the database schema, since this may make related queries invalid in the programs. Instead, they most probably prefer to add a new table, by duplicating the data and incrementally updating the programs in order to use the new table instead of the old one. In some cases, the old table version is never deleted even when it is not accessed anymore by the programs. Further investigations are needed to confirm this hypothesis.

Roadmap

In this chapter, we have presented a historical analysis approach that allows us to analyze, at a fine-grained level, how the program source code and the database schema have co-evolved over time. This historical analysis relies on a data model gathering information about the evolution history of several system artefacts (i.e., source code, database schema, database usage and ORM usage).

In Chapters 6 and 7, we exploit this data model; Chapter 6 presents a tool-supported approach, that allows developers to 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. Chapter 7 presents DAHLIA 2.0, a visualization tool, that allows developers to analyze the database usage in dynamic and heterogeneous systems by visualizing the interactions between the application program and the database. The proposed visualization uses a city metaphor.

CHAPTER 6

DETECTING AND PREVENTING PROGRAM INCONSISTENCIES UNDER DATABASE SCHEMA EVOLUTION

Based on the historical analysis presented in the previous chapter, this chapter[a] *presents a tool-supported approach that allows developers to 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.*

[a]This chapter is an extension of our paper [Meurice et al., 2016b] published in the proceedings of the International Conference on Software Quality, Reliability and Security.(QRS 2016) and was granted the *Best Paper Award*.

6.1 Introduction

We observed in previous chapters that data-intensive applications tend to access their underlying database in an increasingly dynamic way. This level of dynamicity significantly complicates the task of adapting application programs to database schema changes. 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. Indeed, failing to correctly adapt programs to an evolving database schema results in program inconsistencies, which in turn may cause program failures.

This Chapter 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.

6.2 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 behaviour 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.

6.2.1 Analysis Background

For achieving this, we will analyze the history of three large real-life systems (i.e., OSCAR, OpenMRS and Broadleaf) 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 [Sjøberg, 1993; Curino et al., 2008; Vassiliadis et al., 2015], 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 [Curino et al., 2008], 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.

For analyzing the co-evolution history of each system, we exploited their corresponding database (produced in Section 5.4.1) which contains information about the evolution history of each system.

6.2.2 Co-Evolution History Analysis

Analyzing how database schemas and programs co-evolve in Broadleaf, OpenMRS and OSCAR will help us to establish the usefulness and the potential benefit of our what-if analysis approach which can support developers to achieve co-evolution tasks (the what-if analysis will be presented later in Section 6.3). For that, we analyze the historical databases 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.

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 6.1 and 6.2 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 average time (expressed in number of versions) needed to adapt the code (no more ORM mapping or access to the table/column). 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. 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.

System	#Table Deletions	#Unsolved	Propagation Time avg~max	#Accesses avg~max
OpenMRS	11	1	13.1 ~ 134	7.5 ~ 9
Broadleaf	86	0	1.1 ~ 6	2.8 ~ 14
OSCAR	33	5	6.6 ~ 90	4.1 ~ 9

Table 6.1: Co-Evolution metrics related to table deletions.

System	#Column Deletions	#Unsolved	Propagation Time avg~max	#Accesses avg~max
OpenMRS	32	4	1.7 ~ 7	2.2 ~ 4
Broadleaf	154	0	1.1 ~ 2	4 ~ 15
OSCAR	170	0	1.1 ~ 24	6.6 ~ 132

Table 6.2: Co-Evolution metrics related to column deletions.

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 history, 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.

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 forgot 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 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 not all developers are 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

[1]The reader can find in (http://bit.ly/1XzyuWu) the proof of the existence of that mapping in April 2015. The database schema of that version can be found in (http://bit.ly/24aa3DQ).

[2]The `VOIDED` column has been removed from the `USERS` table in September 2011 and is still currently accessed (http://bit.ly/1Q39ipQ) 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 (http://bit.ly/1OhJYqu) and defined a Hibernate mapping until April 2014 (http://bit.ly/1Q39but).

System	Renaming	Solution Time avg~max	#Accesses avg~max
OpenMRS	1	1 ~ 1	0 ~ 0
Broadleaf	14	2.9 ~ 6	3.3 ~ 8
OSCAR	7	13.6 ~ 89	4.1 ~ 9

Table 6.3: Co-Evolution metrics related to table renamings.

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 thesis.

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. The average time to propagate the table deletions is quite significant (6.6 versions), with a maximal value of 90 versions; it illustrates 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.

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 6.3 and 6.4 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.

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

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	34.9 ~ 389

Table 6.4: Co-Evolution metrics related to column renamings.

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 4.1 impacted locations per table renaming (and one maximal value of 9 locations). However, OSCAR developers have renamed the columns considerably more, with 321 column renamings. On average, 34.9 locations are impacted by a column renaming, with an extreme value of 389 locations. 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, 2.9 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].

Discussion

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,

[4]The `FORMCOUNSELLORASSESSMENT` table was renamed in December 2006 but was still accessed in September 2010 (http://bit.ly/1RkAn7w).

[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 (http://bit.ly/1VpsZIF) and after (http://bit.ly/1oMgEE3) the renaming.

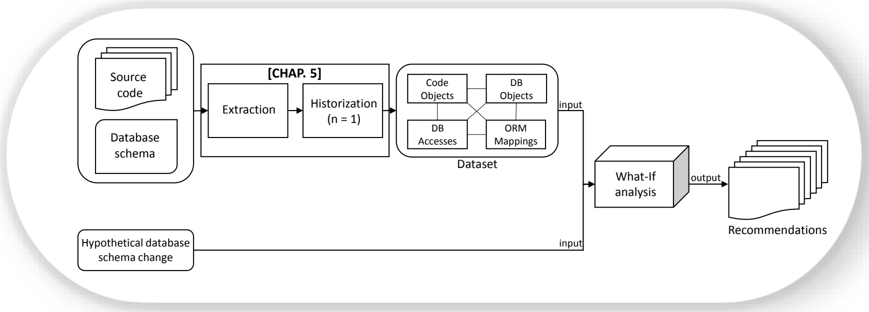


Figure 6.1: Overview of our what-if analysis approach.

developers keep creating new accesses to removed or renamed schema objects. It allowed us to point out query failures related to awkward past changes.

6.3 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.

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 perform this particular database schema change?".

Figure 6.1 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 apply the extraction and historization phases as described in Chapter 5. The historization phase is applied to a single system version (n=1) and therefore, the notion of version present in the ER model (Figure 5.13) becomes useless. 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 [Sjøberg, 1993; Curino et al., 2008; Vassiliadis et al., 2015], 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 [Curino et al., 2008], 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 6.5. 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 6.2 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 6.6). The developers cope with two database schema changes:

(a) 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**)

(b) 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**).

[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`.

Operation	Strategy	Type	ID
Deleting table	• Deleting all the ORM mappings of M_o • Modifying/Deleting all the accesses of A_o^e	Recommendation Recommendation	I 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. • 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 Recommendation	III IV
Deleting column	• Deleting all the ORM mappings of M_o • Modifying/Deleting all the explicit accesses of A_o^e. • 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.	Recommendation Recommendation Warning	I II V
Renaming column	• Modifying the column in each mapping of M_o • Modifying all the SQL queries of A_o^e • 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.	Recommendation Recommendation Warning	III IV V
Changing column type	• Changing the type of all mapped attributes in M_o • 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. • 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 Warning Warning	VI VII V

Table 6.5: Strategies (recommendations and warnings) for facing a database schema change.

	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 6.6: The code locations accessing the `CUST` table and the `POSTAL_CODE` column detected by our what-if analysis approach.

```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;
  }
}
```

Figure 6.2: Java code before (left) and after (right) the co-evolution with the help of our what-if analysis approach.

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

Table 6.7: Distribution of the 130 selected database schema changes.

6.4 Evaluation

In this Section, we assess the accuracy of our what-if analysis approach. This evaluation aims to measure (1) *correct 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 6.5 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 6.7 shows the distribution of those changes.

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:

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

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

(c) *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**

	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%**

Table 6.8: Rates of correct, wrong and missing recommendations.

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 6.8 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.

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.

6.5 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.

```
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();
```

Listing 6.1: 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.

Database Access Extraction

In our co-evolution history analysis (Section 6.2) and what-if analysis (Section 6.3), we use our historical analysis approach proposed in Chapter 5 which, in turn, builds on our static analysis approach (presented in Chapter 4) 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 (discussed in Section 4.3.3), which may affect our results:

- 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). Listing 6.1 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.
- Missing queries: some queries cannot be fully recovered by our analyzer due to its static nature. In Section 4.3.3, 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 Section 4.3: we could extract queries for 71.5%-99% of database accesses with 87.9%-100% of valid queries.

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 perform the database schema historization phase (detailed in Section 5.2.2), we compare successive database schema versions. However, 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, 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 Section 3.2.5. 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.

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 most likely database schema changes 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.).

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.

6.6 Concluding Remarks

In this Chapter, 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 a 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.

Heterogeneity: an anti-pattern?

In Section 6.2, we analyzed the behaviour of developers (from three particular systems) when adapting programs to database schema changes. We studied how source code and database co-evolved over time. Based on our observations, we can question the suitability to have heterogeneity within a system. Several reasons can explain heterogeneity:

(a) **Developers' preferences**: according to her/his preferences, a developer can be more comfortable with a particular access technology. In large development teams, one could therefore observe the use of several technologies. Moreover, the high turnover in teams and the arrivals of young developers in the project can also encourage the introduction of new popular technologies.

(b) **Historical reasons**: as observed in the OSCAR project, heterogeneity within a project can be due to historical reasons. Initially, OSCAR only used JDBC to access its database. Later, with the increasing popularity of ORM technologies, Hibernate was introduced to complement JDBC. Later again, we observed the introduction of JPA in order to fully replace Hibernate - developers seemed to prefer the use of JPA annotations to map code objects to database objects instead of XML configuration files. However, this attempt of migration failed and today, the three technologies still co-exist.

(c) **Using a technology to complement another one**: whereas using ORM technologies (e.g., JPA/Hibernate) offer a lot of advantages, they are not always sufficient to replace JDBC. In particular cases, ORM does not allow some queries which are supported by JDBC. For instance, Hibernate does not support the insertion of multiple objects in a same table by using a single query; instead, developers have to write separate queries to insert each object. Or again, for some complex data, using ORM technologies instead of JDBC can reduce performance. Hence, using JDBC as complement to an ORM technology can be sometimes indispensable to palliate its drawbacks.

Using an ORM technology to access the underlying database can have several advantages[7]; with JDBC, developers have to write code to map an object model's data representation to a relational data model. Thus, with JDBC, developers have to manually manage the evolution of those mapped objects in case of schema changes. ORM offers a flexible and powerful solution to map code objects to database objects. In case of minor schema modifications, developers only need to change the defined ORM mappings, reducing the development time and maintenance cost. In addition, ORM also allows developers to avoid writing complex SQL queries; instead, ORM provides simple and effective ways to perform a database manipulation task. However, as discussed above, JDBC can also palliate some ORM drawbacks; hence, the use of both technologies, i.e., ORM and JDBC, can be useful.

Although our study revealed that the most heterogeneous system (i.e., OSCAR) was the most *difficult to maintain*, and that the less heterogeneous system (i.e., Broadleaf) was the most *maintainable* in terms of propagation time (see Tables 6.1, 6.2, 6.3 and 6.4), we truly think that using several access technologies within the same system can sometimes be a good solution, **if developers are sufficiently disciplined and properly trained**. For instance, using JDBC, as complement to an ORM technology (e.g., Hibernate or JPA) can constitute a good solution. Indeed, ORM allows reducing the maintenance cost and JDBC can palliate some ORM drawbacks. However, the price developers have to pay for heterogeneity is to be properly disciplined and prepared, in order to deal with several access technologies at once, in the context of program-database co-evolution.

[7]http://www.mindfiresolutions.com/mindfire/Java_Hibernate_JDBC.pdf

Roadmap

In this chapter, we have presented a what-if analysis approach giving developers recommendations on how and where to propagate a hypothetical schema change to the programs. We finally evaluated our approach and obtained very promising results.

In the next chapter (Chapter 7), we present DAHLIA 2.0, a 3D visualization tool, that allows developers to analyze the database usage in dynamic and heterogeneous systems by visualizing the interactions between the application program and the database.

CHAPTER 7

VISUAL ANALYSIS OF DATABASE USAGE IN DYNAMIC AND HETEROGENEOUS SYSTEMS

In this chapter[a]*, we present DAHLIA 2.0, an extension of the visualization tool introduced in Section 3.3.6. DAHLIA 2.0 allows developers to analyze the database usage in dynamic and heterogeneous systems by visualizing the interactions between the application program and the database. We secondly apply DAHLIA 2.0 to real-life systems and illustrates the benefits of this visualization. Finally, we present a controlled experiment for the empirical evaluation of DAHLIA 2.0; the objective is to experimentally prove the suitability of our approach in the context of program-database co-evolution.*

[a]This chapter is an extension of our tool demo paper [Meurice and Cleve, 2016] published in the proceedings of the 4th IEEE Working Conference on Software Visualization (VISSOFT 2016).

7.1 Introduction

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. However, data-intensive applications nowadays 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 ORM frameworks. This level of dynamicity significantly complicates the task of adapting programs to an evolving database schema. Indeed, we previously observed that manually recovering the links between the source code and the database schema and understanding it may prove complicated due to these higher levels of abstraction and dynamicity.

In this Chapter, we present DAHLIA 2.0, a visualization tool allowing developers to analyze the database usage in highly dynamic and heterogeneous systems. DAHLIA 2.0 supports software comprehension and database-program co-evolution.

7.2 Visualization

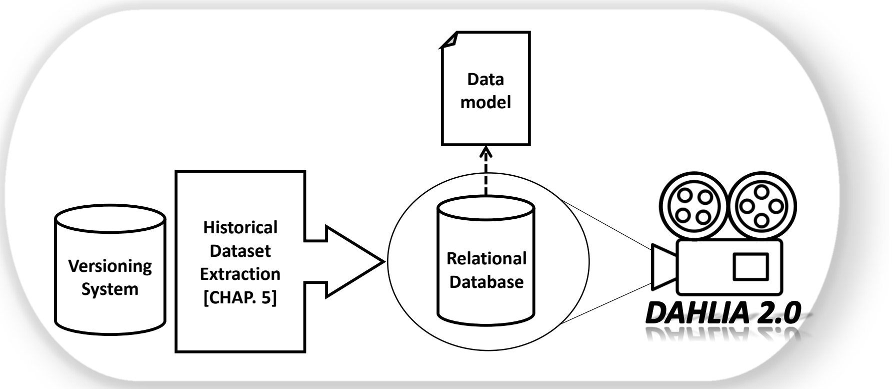


Figure 7.1: The DAHLIA 2.0 architecture. DAHLIA 2.0 takes as input the historical dataset, extracted as described in Chapter 5. This historical dataset must be stored in a relational database implementing the data model presented in Section 5.2.2. The database structures are defined in Section 5.3.2.

DAHLIA 2.0, whose architecture is depicted in Figure 7.1, requires as input the historical dataset derived by our approach presented in Section 5.2. This dataset must be stored in a database structured as we detailed in Section 5.3.2. The visualization tool queries this database in order to compute and visualize information related to the database usage of a subject DISS at a given version[1].

We extended our visualization tool DAHLIA 1.0 to allow developers to analyze the database usage of a system. While DAHLIA 1.0 only considers the database schema history, DAHLIA 2.0 is now able to visualize the database usage by exploiting the links between the program source code and the database. The main role of DAHLIA 2.0 is to provide developers with a visual support to database-program co-evolution by analyzing the dependencies between the code and the database; such a visualization can help assessing the costs of a future system change (e.g.,

[1]Each of those features can be applied to any considered system versions; the latest version as well as any older one.

what would the code locations to maintain be if I delete this database table?). We list below some of the novel features implemented in DAHLIA 2.0[2].

7.2.1 Visualizing the Database City

This feature reuses the 3D city-metaphor proposed by Wettel *et al.* [Wettel and Lanza, 2008a] that facilitates the visualization of very large database schemas. A database table is represented as a 3D building. We use the building height, width and color for representing database usage metrics. The user may select the metrics to affect to each dimension/property and may thus customize the city according to his/her needs. Table 7.1 summarizes the visual mappings affected to the *database* city.

(a) Building height: the building height is mapped to the number of columns. This visual metric gives indications about the table size.

(b) Building width: the building width can be mapped to either (1) the number of database queries sent from the source code that access the table, or (2) the number of code locations accessing the table. These metrics give indications about the table manipulation within the source code. It provides an estimation of the required effort to co-evolve the code in case of future table modifications.

(c) Building color: a color is affected to each building. A building can be colourized according to either (1) the technologies accessing the table (e.g., a particular color for all the database tables accessed by a given technology), or (2) the ORM mappings defined on the table (e.g., a particular color for all the database tables that are mapped to the code via an ORM). Those visual metrics give indications about the technological manipulation of each table.

That kind of metrics permits developers to instinctively detect the *sensitive* database parts intensively linked (accessed) to the code. The user is free to affect any *width* metric to the building *height* and vice-versa. Table 7.1 only represents a "recommended" metric assignment.

Figure 7.2 depicts an example of 3D database city that one can visualize within DAHLIA 2.0. In this example, each building (table) has a height denoting the number of columns, a width representing the number of accessing queries sent from the source code, and a color representing the database access technology. As illustrated, the user can visualize the corresponding 2D table form.

7.2.2 Visualizing the Code City

That feature proposes a visualization of the program source code similar to [Wettel and Lanza, 2008a] by representing a file as a 3D building. The novelty we propose

[2]All the features of DAHLIA 1.0 are included in DAHLIA 2.0. We only present in this Section the new features. The former ones are detailed in Section 3.3.6.

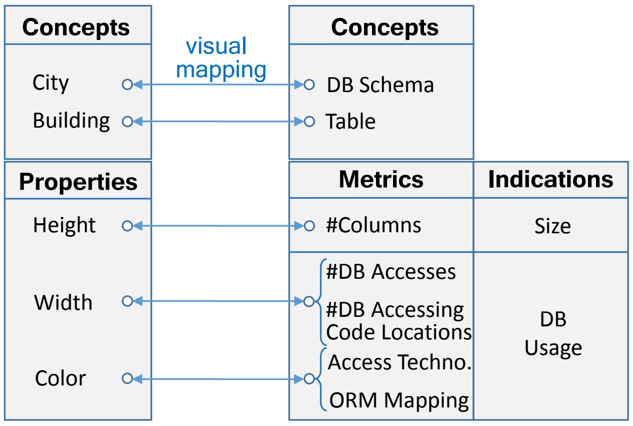
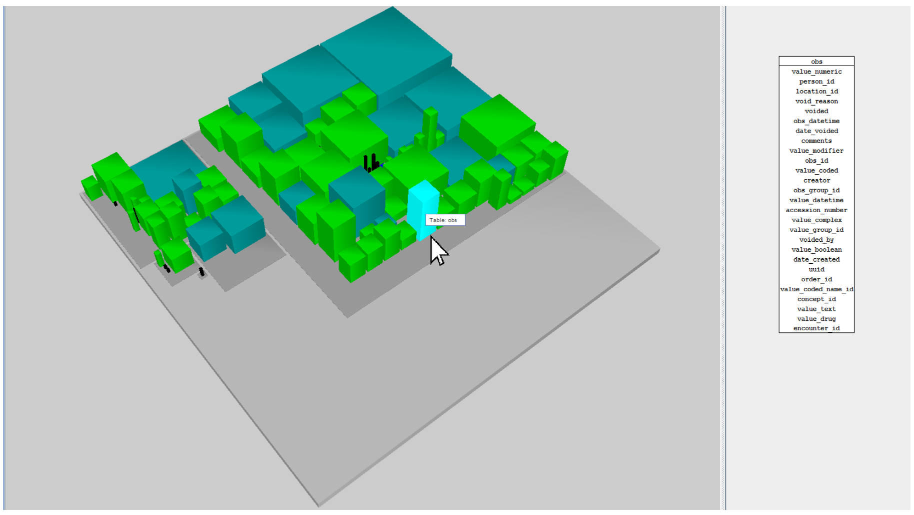


Table 7.1: The visual mappings affected to the database city.



```
obs
value_numeric
person_id
location_id
void_reason
voided
obs_datetime
date_voided
comments
value_modifier
obs_id
value_coded
creator
obs_group_id
value_datetime
accession_number
value_complex
value_group_id
voided_by
value_boolean
date_created
uuid
order_id
value_coded_name_id
concept_id
value_text
value_drug
encounter_id
```

Figure 7.2: A 3D database city as visualized within DAHLIA 2.0. The right panel shows the 2D form of a selected table.

is also into the metrics affected to the building height, width and color. Table 7.2 summarizes the visual mappings affected to our *code* city.

(a) Building width: the building width can be mapped to either the number of lines of code within the given file, or the number of methods/functions defined in the given file. Those metrics give indications about the size of each file.

(b) Building height: the building height can be mapped to either (1) the number of queries sent from the given file, (2) or the number of locations accessing the database in the given file, or (3) the number of tables accessed by the given file. Those metrics give indications about the database usage within the source code; it permits the intuitive detection of files intensively linked to the database.

(c) Building color: a color is affected to each building. A building can be coloured according to either (1) the database access technologies used within the given file (e.g., a particular color for the files using a given database access technology), or (2) the ORM mappings defined in the given file (e.g., a particular color for the files defining some ORM mappings). Those metrics give indications about the technological distribution in the source code.

Here again, that kind of metrics will allow the immediate detection of *sensitive* code parts intensively linked to the database. The visual metrics can be chosen by the user. Moreover, the latter is free to affect any *width* metrics to the building *height* and vice-versa. Table 7.2 only represents a "recommended" metric assignment.

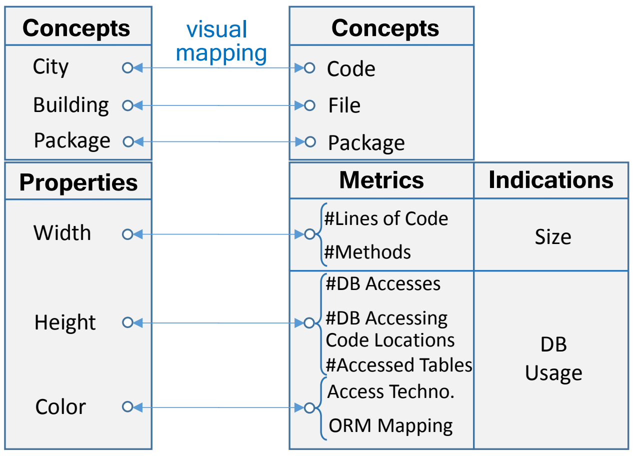
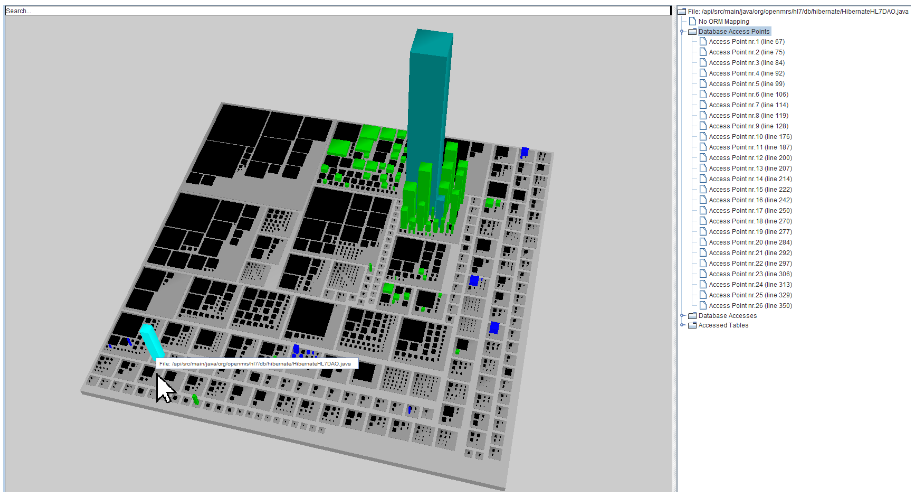


Table 7.2: The visual mappings affected to the code city.

Figure 7.3 shows an example of 3D code city as visualized within DAHLIA 2.0. Each building - a Java file in that case - has a height, i.e., the number of locations accessing the database), a width, i.e., the number of methods in the file, and a color for the database access technology(ies) used in the file (black = none, green = Hibernate, blue = JDBC, mix = JDBC & Hibernate).

7.2.3 Visualizing the links between the Database and Code city

This feature proposes a dual visualization; the database and code cities are displayed side-by-side according to the metrics chosen by the user. It enables the user to assess the costs of a future database schema change or a code refactoring step. The user can click on a particular file and visualize which part of the database schema is accessed. The results of a click on a file will be (1) the highlighting of all the


Search...
File: /api/src/main/java/org/openmrs/hl7/db/hibernate/HibernateHL7DAO.java

```
File: /api/src/main/java/org/openmrs/hl7/db/hibernate/HibernateHL7DAO.java
  No ORM Mapping
  Database Access Points
    Access Point nr.1 (line 67)
    Access Point nr.2 (line 75)
    Access Point nr.3 (line 84)
    Access Point nr.4 (line 92)
    Access Point nr.5 (line 99)
    Access Point nr.6 (line 106)
    Access Point nr.7 (line 114)
    Access Point nr.8 (line 119)
    Access Point nr.9 (line 128)
    Access Point nr.10 (line 176)
    Access Point nr.11 (line 187)
    Access Point nr.12 (line 200)
    Access Point nr.13 (line 207)
    Access Point nr.14 (line 214)
    Access Point nr.15 (line 222)
    Access Point nr.16 (line 242)
    Access Point nr.17 (line 250)
    Access Point nr.18 (line 270)
    Access Point nr.19 (line 277)
    Access Point nr.20 (line 284)
    Access Point nr.21 (line 292)
    Access Point nr.22 (line 297)
    Access Point nr.23 (line 306)
    Access Point nr.24 (line 313)
    Access Point nr.25 (line 329)
    Access Point nr.26 (line 350)
  Database Accesses
  Accessed Tables
```

Figure 7.3: A 3D code city as visualized within DAHLIA 2.0. The right panel shows information about a selected file, i.e., the access locations, the accessed tables and the ORM mappings.

accessed tables and (2) the accurate detection of all the code locations accessing them. Figure 7.4 depicts the database and code cities as visualized within DAHLIA 2.0. The database (left) and code (right) cities are side-by-side. The green tables are tables with Hibernate mapping, black tables are tables without any ORM mappings. The table height represents the number of columns while the table width is the number of SQL queries accessing it. The green files are files using Hibernate, blue files are files using JDBC and black files do not access the database. The file height represents the number of accessed tables while the width represents the number of locations accessing the database. Figure 7.4 illustrates the following scenario: the user plans to refactor the `HibernateConceptDAO` Java file and wishes to assess the costs of that evolution phase with help of DAHLIA 2.0. The user clicks on the `HibernateConceptDAO` file (highlighted building in the right city depicted in cyan). DAHLIA automatically and instantly highlights (cyan color) all the database tables (left city) accessed by that file. By this way, the user can directly have an estimation of the required effort to perform that evolution phase.

The reverse operation is also possible: the user can click on a particular table in order to highlight the files accessing the selected table. It provides an assessment of the impact of a future table modification.

7.2.4 Jumping into the code

In addition to the 3D support, the tool can list, on its right panel, a large set of information. For instance, the user can decide to list (1) all the tables accessed by a given file, (2) all the precise code locations (precision in terms of line of code) in a given file which allows accessing the database, (3) all the (SQL) queries accessing a given

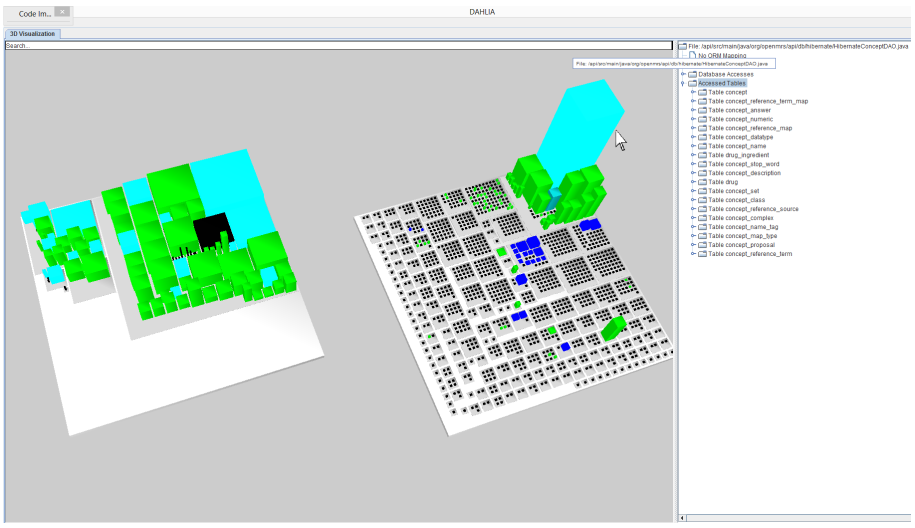


Figure 7.4: Database (left) and Code (right) cities side-by-side as visualized within DAHLIA 2.0. The right panel shows information about a selected file.

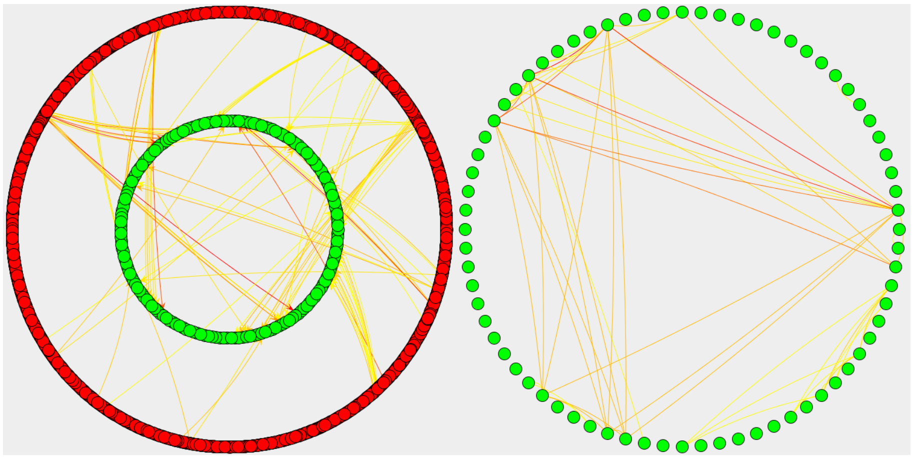
table (as well as their actual value, the accessed database objects and their execution location in the code), etc. Thanks to this information panel, the user has an accurate report on the database usage. Moreover, that panel allows the user to directly jump into the precise code locations that will require some modifications/adaptations in case of code/database change.

7.2.5 Visualizing the database-program dependencies within a circular view

This feature uses a circular visualization. Unlike the 3D visualization, this mode allows to directly visualize the dependencies between the program source code and the database in terms of intensity. Precisely determining the dependencies between a table and a file or between a table and another table can considerably help the user to assess the future impact of any change on the system. Figure 7.5 depicts two examples of circular visualization.

(a) Visualizing the dependencies between the source code files (represented in red) and the database tables (represented in green) in terms of access locations. The higher the number of locations in the file accessing the table, the darker the color.

(b) Visualizing the dependencies between the tables themselves in terms of closeness; namely, two tables are close when they appear together in a same SQL query. The higher the number of queries where the two tables appear together, the closer they are and thus, the darker the color.



(a) Dependencies between files and tables. (b) Dependencies between tables.

Figure 7.5: Circular views as visualized within DAHLIA 2.0. Red bullets represent files, while green ones depict tables. Darker the color stronger the dependency.

System	Description	LOC	Tables	Accesses
Bfit	Library content management system	32 839	38	87
Broadleaf	E-commerce framework	254 027	179	930
DAHLIA	Visualization tool	124 888	26	273
Liferay	Enterprise web platform	2 398 861	145	4 617
MusicBrainz	Encyclopedia of music information	4 097	286	70
Oopms	Online project management suite	260 536	188	1 604
OpenEMM	Web-based enterprise application	102 529	68	9 162
OpenMRS	Medical record system	301 232	88	951
OSCAR	Medical record system	2 054 940	480	13 822
QuanLyVatTu	Commerce management system	13 589	17	244
Sgaf	Billing management system	15 440	50	112

Table 7.3: Size metrics of the studied systems.

Moreover, the user can decide to only select a particular table/file and display its dependencies with the other objects.

7.3 Application to Real-Life Systems

We used DAHLIA 2.0 to visualize 11 real-life systems. Table 7.3 gives some metrics pertaining to the studied systems. In terms of lines of code, the size of these system varies: we selected small (< 10KLOC), medium and large systems (> 2000KLOC).

Figures 7.7, 7.8, 7.10, 7.9, 7.11, 7.12, 7.13, 7.14, 7.15, 7.16 and 7.17 depict the dual visualization (database and code cities) with DAHLIA 2.0 of, respectively, Bfit[3], Broadleaf[4], DAHLIA itself, Liferay[5], MusicBrainz[6], Oopms[7], OpenEMM[8], OpenMRS[9], OSCAR[10], QuanLyVatTu[11] and Sgaf[12].

The database city (left city) uses several metrics. The building height represents the number of columns; the building width represents the number of code locations accessing the table; the building color represents the technology(ies) accessing the table (the color is affected according to the legend depicted in Figure 7.6).

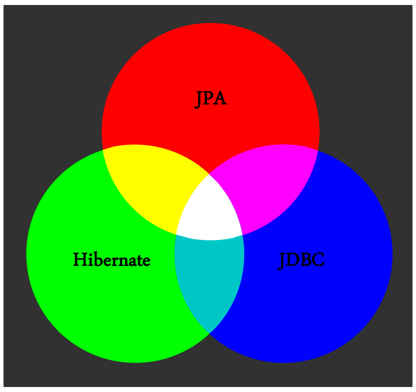


Figure 7.6: Legend of the colourization used within DAHLIA 2.0. Each technology and intersection of technologies has a particular color. The objects not concerned by any technologies are depicted in black.

The code city (right city) uses several metrics. The building height represents the number of code locations within the file accessing the database; the building width represents the number of lines of code; the building color represents the database access technology(ies) used within the file (the color is affected according to the legend depicted in Figure 7.6).

Each system has its own characteristics and architecture.

- Bfit: Bfit only uses Hibernate as database access technology. The majority of tables (left city) are manipulated/accessed within the source code; only a few ones remain unaccessed (black tables). Concerning the code (right city), all the database-related files are into the same package, which decreases the time search to locate the accessing code locations in case of database schema modifications.

[3] https://github.com/yalelibrary/bulk-fcrepo-import (date:24/03/2016)
[4] https://github.com/BroadleafCommerce/BroadleafCommerce (date:31/03/2015)
[5] https://github.com/liferay/liferay-portal (date:12/06/2010)
[6] https://github.com/lastfm/musicbrainz-data (date:31/08/2015)
[7] https://github.com/Ankithukral/oopms (date:24/08/2012)
[8] https://sourceforge.net/projects/openemm/files/ (date:18/11/2009)
[9] https://github.com/BroadleafCommerce/BroadleafCommerce (date:10/04/2015)
[10] https://github.com/scoophealth/oscar/ (date:14/12/2014)
[11] https://github.com/free-development/QuanLyVatTu (date:07/10/2015)
[12] https://github.com/organizationFreeLance/sgaf (date:12/06/2016)

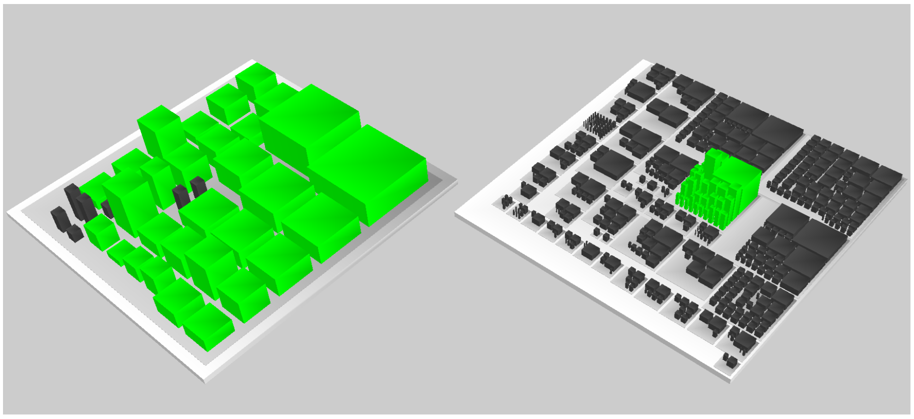


Figure 7.7: Visualization of Bfit with DAHLIA 2.0.

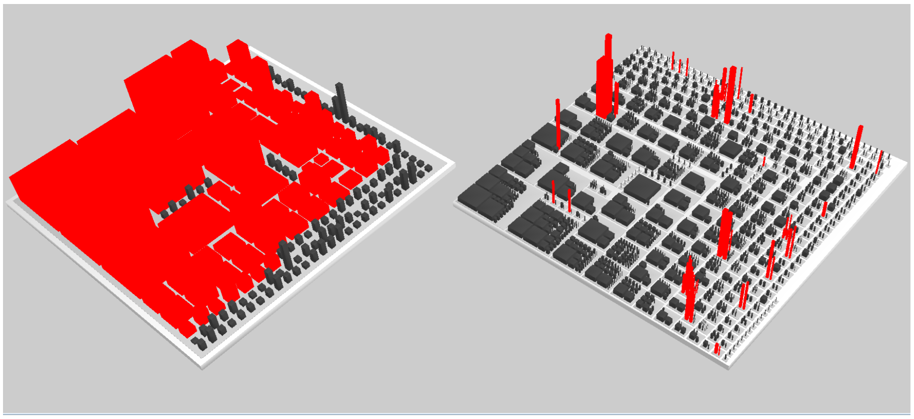


Figure 7.8: Visualization of Broadleaf with DAHLIA 2.0.

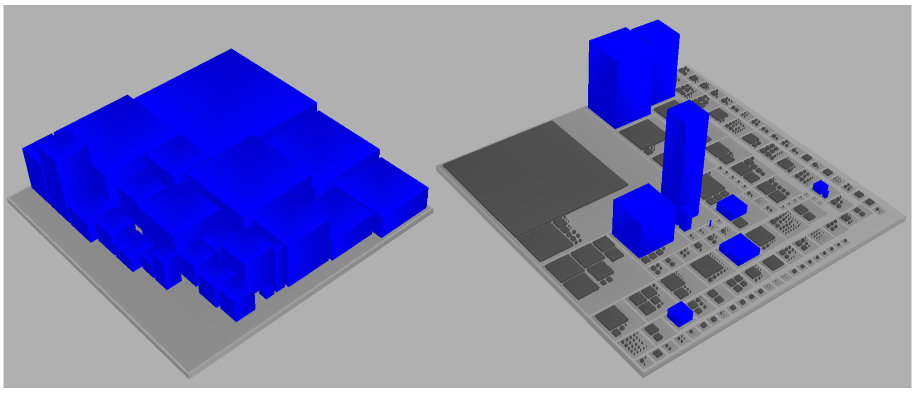


Figure 7.9: Visualization of DAHLIA with DAHLIA 2.0.

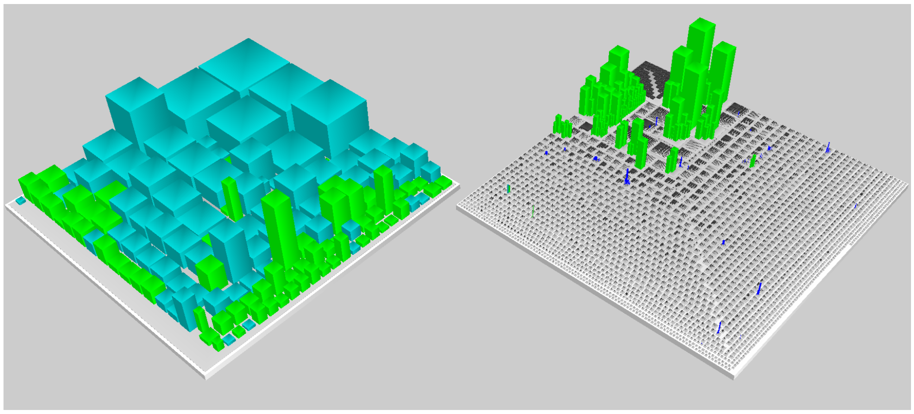


Figure 7.10: Visualization of Liferay with DAHLIA 2.0.

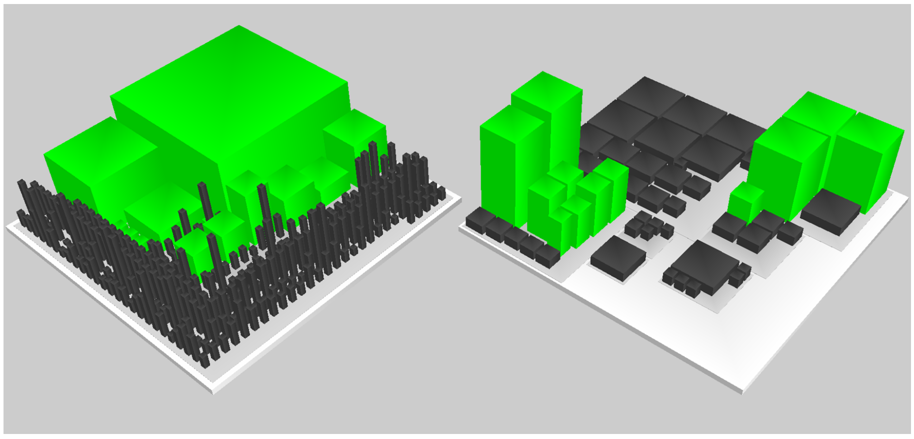


Figure 7.11: Visualization of MusicBrainz with DAHLIA 2.0.

- Broadleaf: Broadleaf only uses JPA to access its database. One notices that a lot of tables are unaccessed. Concerning the code, its organization seems in disorder with dozens of packages containing (sometimes isolated) database access files.
- DAHLIA: DAHLIA only uses JDBC to access its database. All the tables are accessed within the source code. Concerning the source code, only a dozen of files manage the communication with the database. One observes that the majority of the access points are located in three main files.
- Liferay: Liferay uses Hibernate and JDBC together to access its database. However, one notices that Hibernate is the dominant technologies in terms of accessed tables and database access code locations. It seems that JDBC is used to complement Hibernate. One also observes a certain discipline in the database-related source code files; there is no "mixed" files containing both Hibernate and JDBC locations. JDBC and Hibernate parts remain separated.

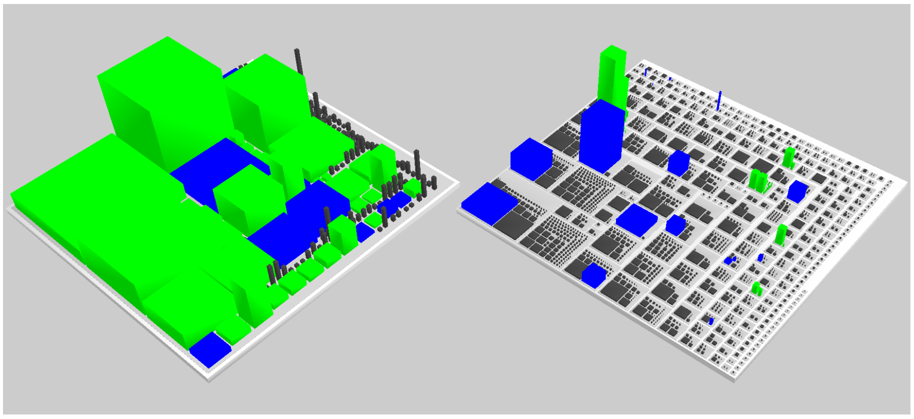


Figure 7.12: Visualization of Oopms with DAHLIA 2.0.

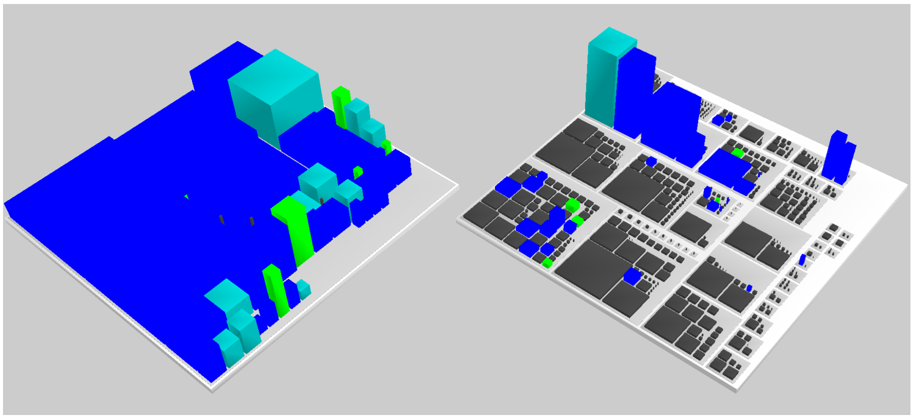


Figure 7.13: Visualization of OpenEMM with DAHLIA 2.0.

It is also interesting to observe that the largest files, in terms of lines of code, are the database access files. Moreover, some database access files are quite massive in terms of number of access locations (building height); detecting those massive files allows us to spot the sensitive files, intensively linked to the database.

- MusicBrainz: MusicBrainz is a small-size system. It uses Hibernate as unique access technology. The majority of the tables are unaccessed; this is explained by the fact that MusicBrainz is designed to manage an encyclopedia of music information. This encyclopedia, stored in a relational database, is used by several other applications - each one for a specific objective. MusicBrainz is meant for read-only access to this database and only uses a subset of the database schema.
- Oopms: Oopms uses Hibernate and JDBC to access its database. However, both technologies are totally separated in the source code files and packages

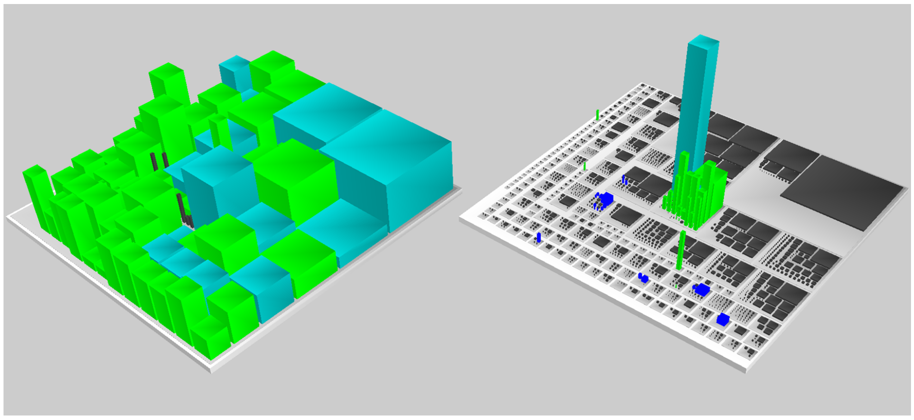


Figure 7.14: Visualization of OpenMRS with DAHLIA 2.0.

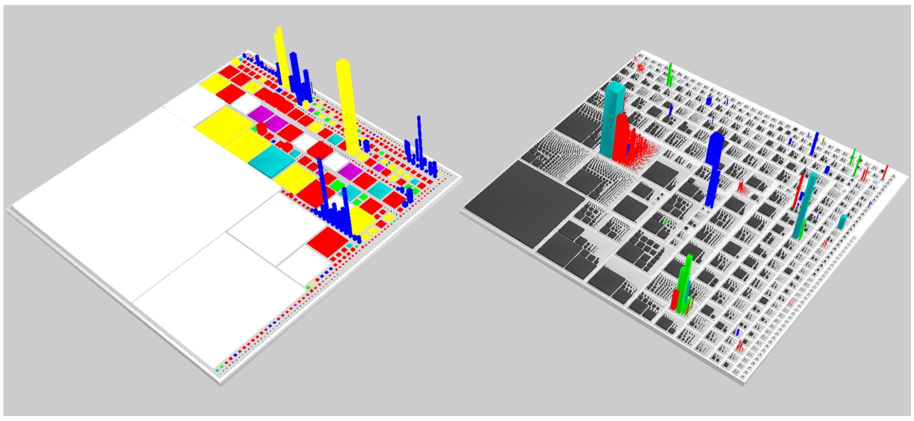


Figure 7.15: Visualization of OSCAR with DAHLIA 2.0.

(no mixed files containing both technologies). The same trend is observed concerning the technological distribution across the schema; each table is accessed by, at most, one technology. Moreover, one observes that among the largest files, some of them are database access files. In general, one sees that the database access files are quite scattered, with very few of them in a same package.

- OpenEMM: OpenEMM uses JDBC and Hibernate as access technologies. There is only a few unaccessed tables. It seems that JDBC is the main used technology in terms of technology distribution across the database schema as well as the source code files. However, one can observe the presence of a "mixed" file using both Hibernate and JDBC to query the database. This kind of observations might lead to a first awareness that refactoring this file could be recommended.
- OpenMRS: OpenMRS makes also use of Hibernate and JDBC. Almost all the

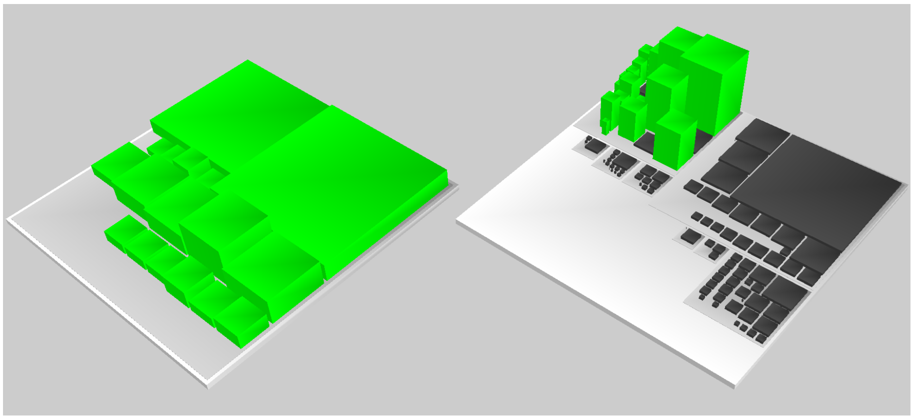


Figure 7.16: Visualization of QuanLyVatTu with DAHLIA 2.0.

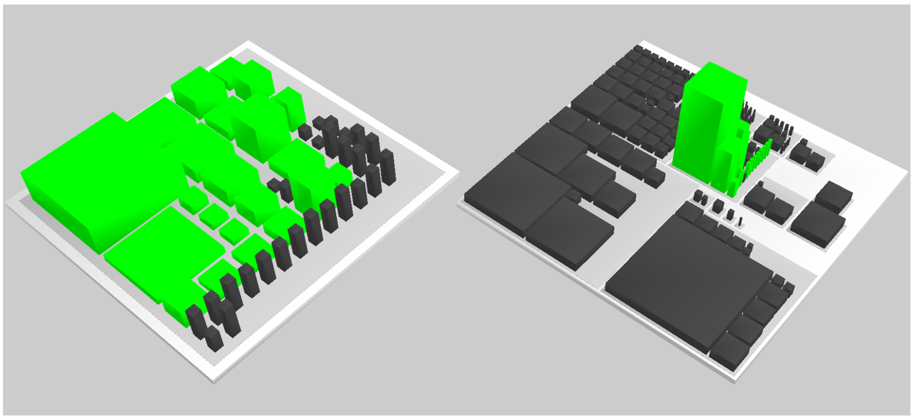


Figure 7.17: Visualization of Sgaf with DAHLIA 2.0.

tables are accessed in the code. Moreover, some of them are accessed by both technologies. Similarly, one observes the existence of a "mixed" file using JDBC and Hibernate. Here again, this detection might be a signal for the necessity of refactoring. Furthermore, this file has an important number of access code locations, which motivates it again. Unless the code size, the database access files are restricted to a limited number of packages, which proves the presence of a certain order in the code organization.

- OSCAR: OSCAR uses together JDBC, Hibernate and JPA to access its database. As discussed in Section 5.4.2, the presence of the three technologies is due to historical reasons and failed attempts to migrate from a technology to another. Because of those reasons, the OSCAR architecture is in complete disorder. Some tables are accessed by several technologies - sometimes, the three technologies! - and this disorder is mainly observable within the source code itself: some packages and files contain several access technologies, which clearly depicts a certain lack of discipline in the code organization. One can also detect some huge files in terms of access code locations. This dual visualization of the OSCAR architecture proves the growing unmaintainability of the system.

- QuanLyVatTu: QuanLyVatTu only accesses its database via Hibernate. This system is a small-size system. Moreover, all the database access files are into the same package and each of them has a relatively low number of access locations.
- Sgaf: Sgaf has an architecture very similar to QuanLyVatTu, using Hibernate as only access technology. The only difference is the important number of unaccessed tables. All the database access files are into the same package.

7.4 Evaluation

In this Section, we present a controlled experiment that empirically evaluates DAHLIA, our visualization approach. The objective is to experimentally prove the suitability of our approach in the context of program-database co-evolution. The experiment quantitatively measures how DAHLIA 2.0 influences (1) the time required to achieve typical comprehension tasks pertaining to the program-database communication, and (2) the correctness of the answers related to those comprehension tasks.

7.4.1 Hypothesis Formulation

We define two research questions underlying our experiment:

- RQ1: Does DAHLIA 2.0 reduce the time needed to complete the tasks?
- RQ2: Does DAHLIA 2.0 increase the correctness of the tasks?

The null hypotheses corresponding to those two research questions are formulated as below:

- $H0_1$: there is no difference in the completion time for different tasks between participants using DAHLIA 2.0 and those ones who do not use DAHLIA 2.0.
- $H0_2$: there is no difference in the correctness of responses between participants using DAHLIA 2.0 and those ones who do not use DAHLIA 2.0.

Therefore, the alternative hypotheses are formulated as below:

- $H1_1$: there is a difference in the completion time for different tasks between participants using DAHLIA 2.0 and those ones who do not use DAHLIA 2.0.

- $H1_2$: there is a difference in the correctness of responses between participants using DAHLIA 2.0 and those ones who do not use DAHLIA 2.0.

7.4.2 Variable Selection

The purpose of the experiment is to show whether DAHLIA 2.0 provides better support in solving program-database communication comprehension tasks, compared to the baseline. Our experiment has one independent variable: the *tool* used to solve the tasks. In addition, our experiment has two dependent variables, i.e., the *completion time* and the *correctness*.

The choice of a baseline. We looked for a baseline on which we could fairly compare DAHLIA 2.0, unfortunately we could not find a single tool visualizing the program-database communication at a precision level similar to DAHLIA 2.0.
Moreover, CodeCity [Wettel and Lanza, 2008a] cannot represent a fair baseline since it does not handle/visualize the interactions between the database and programs.

In one concern of fairness, we also rejected Eclipse IDE as baseline. Indeed, we are intimately convinced that navigating through the source code with Eclipse to recover and understand the program-database communication is not a conceivable solution; dynamically generated queries and ORM frameworks considerably complicate the task of manually identifying and recovering database accesses within the source code. Therefore, opting for Eclipse as baseline would bias the experiment. Instead, in order to avoid to confer an unfair data advantage to the subjects in the experimental group, we decided to provide the control group with data tables containing the metrics required to solve the tasks. Those data are stored in Excel sheets allowing a simple data exploration. Those Excel sheets are available in Appendix A.

The object system. To evaluate our approach, we selected OpenMRS as object system. We truly think that OpenMRS represents a well-suited real-life system to perform this experiment, in terms of size (> 300 KLOC, 1,426 java files and 88 tables) and used database access technologies (JDBC & Hibernate).

We performed our extraction process as described in Section 5.3.1 and stored the extracted data in a database structured as detailed in Section 5.3.2. From this database, we then automatically fulfilled the Excel sheets with the metrics necessary to solve the experiment tasks.

7.4.3 Tasks Design and Participant Selection

We defined a set of eight comprehension tasks. The tasks are described in Table 7.4. In our case, a task consists of answering a question.

After a first pilot study involving three participants (two PhD students and one post-doc), we conducted the experiment with a total of 48 participants: all participants are students in our university, either in their last year of their Bachelor (B), Master (M) or evening program studies (E). We randomly distributed the students in two groups, i.e., a group of students who will solve the tasks **with** DAHLIA 2.0 and a

Question	Description
Q1	Which technology (JDBC, Hibernate or JPA) accesses the biggest number of database tables?
Q2	How many files use several technologies (>1) to access the database?
Q3	How many files access the database table named *users*?
Q4	How many database tables are accessed by the file named *HibernateHL7DAO.java*?
Q5	How many database tables are accessed by several technologies (>1)?
Q6	How many database tables are not accessed at all?
Q7	Which database table is accessed by the biggest number of code locations?
Q8	Which file accesses the biggest number of database tables?

Table 7.4: Comprehension tasks.

group of students **without** DAHLIA 2.0 (with Excel). Table 7.5 shows the distribution of the participants of each degree in the two groups.

Students	**Groups**	
	DAHLIA 2.0	**Excel**
Bachelor (B)	6	8
Master (M)	6	5
Evening program (E)	13	10
Total	25	23

Table 7.5: Participations distribution.

7.4.4 Data Collection

During our experiment, we collected different data.

Expertise level of Excel. Before the experiment, we collected experience with Excel of each participant belonging to the control group (i.e., using Excel). We defined three possible *levels*: (1) *I never use Excel*, (2) *I have a basic knowledge of Excel* and (3) *I am able to use formulas/functions in Excel.*

Timing Data. To time participants, we implemented our own timer integrated in the questionnaire and running on the participant's computer. We logged the accurate time needed to realize each task. We made the deliberate choice to hide the remaining time in order to avoid stress effects. However, the participant can decide anytime to display the remaining time by clicking on a button. Whenever a participant was unable to finish a task in the allotted time (10 minutes for each task[13]), the application displays a timeout message and invites the participant to pass to the next task (and the timer is then reset).

Correctness Data. To obtain an oracle allowing us to evaluate the correctness of the answers given by the participants, we queried the database used to generate the Excel sheets and used as input of DAHLIA 2.0, and found out the correct answer to each task. Secondly, we manually and blindly - the experimenter did not know if the participant belongs to the control group or not - verified the correctness of each task performed by each participant. We finally graded the participants; the minimum grade is 0, while the maximum is 8 (number of tasks to perform).

The raw measurements of completion time and correctness are available in Appendix A.

7.4.5 Results

Before choosing the suitable statistical test to apply, several assumptions need to be discussed:

(a) **Independence of observations:** the participants were randomly assigned to one of the two groups. The data of each group were independently collected and are therefore *unpaired*. The assumption of independence of observations is thus met.

(b) **Normality of the dependent variable:** we tested the normality of completion time and correctness using the Shapiro-Wilk test [Shapiro and Wilk, 1965]. Based on the null hypothesis H_0 = *data are normally distributed*, we applied the Shapiro-Wilk test to completion time and correctness. We obtained a p-value equal to, respectively, 0.0347 and < 0.0001. Since both are lower than the significant level 0.05, the null hypothesis is rejected.

(c) **Testing outlying observations:** since the hypothesis of a normal distribution is rejected, a second test, i.e., the Grubbs' test [Grubbs, 1950], is required to verify if the null hypothesis H_0 = *there are no outlier data* can be rejected. After applying the Grubbs' test to the completion time and correctness, we obtained a p-value equal to, respectively, 0.097 and 0.310. Since both are greater than the significant level 0.05, the null hypothesis cannot be rejected.

Based on the design of our experiment and our above observations, the suitable non-parametric test for hypothesis testing is the Mann-Whitney test.

After applying the Mann-Whitney test [Mann and Whitney, 1947] to the completion time and the correctness, we obtained a p-value equal to, respectively, <

[13]We set this time limit on the basis of our observations made while conducting the pilot study; we claim that 10 minutes are more than sufficient to answer each question.

	Completion Time (seconds)		Correctness (score)	
	DAHLIA 2.0	**Excel**	**DAHLIA 2.0**	**Excel**
mean	462.68	797.57	7.56	7.04
difference		**-41.99%**		**+7.33%**
min	243	461	6	5
max	809	1361	8	8
median	424	703	8	7
stdev	149.79	219.75	0.71	0.98

Table 7.6: Statistics related to completion time and correctness.

0.0001 and 0.0435. Since both values are lower than the significant level 0.05, our two initial null hypotheses (1) $H0_1$ = *there is no difference in the completion time for different tasks between participants using DAHLIA 2.0 and those ones who do not use DAHLIA 2.0*, and (2) $H0_2$ = *there is no difference in the correctness of responses between participants using DAHLIA 2.0 and those ones who do not use DAHLIA 2.0* can be rejected.

The statistics related to completion time and correctness are presented in Table 7.6.

7.4.6 Result Summary

Completion time. The data allows us to reject the first null hypothesis $H0_1$ in favour of the alternative hypothesis $H1_1$, which states that the tool impacts the time required to complete the tasks. We observed that DAHLIA 2.0 enabled a completion time reduction of 41.99% over Excel. This result is statistically significant.

Correctness. The data also allows us to reject the second null hypothesis $H0_2$ in favour of the alternative hypothesis $H1_2$, which states that the tool impacts the correctness of the solutions to the tasks. We observed that DAHLIA 2.0 enabled an increase in correctness of 7.33% over Excel. This result is statistically significant.

The main effect of the tool on the completion time and correctness is illustrated in Figure 7.18

Task Analysis. Another objective of our experiment was to identify the tasks for which DAHLIA 2.0 provides an advantage over the baseline. Figure 7.19 shows the performance for each task (T), in terms of correctness (the average percentage of correct answer) and completion time (average time to solve the task).

It is not surprising to observe a quite good general correctness for Excel; indeed Excel provides functions (e.g., *sum*, *max*, ...) to find precise answers. Yet, Excel obtains a better correctness with tasks T3 and T4 (even if the correctness for T3 and T4 with DAHLIA 2.0 is good too). The correctness for each other task is better with DAHLIA 2.0.

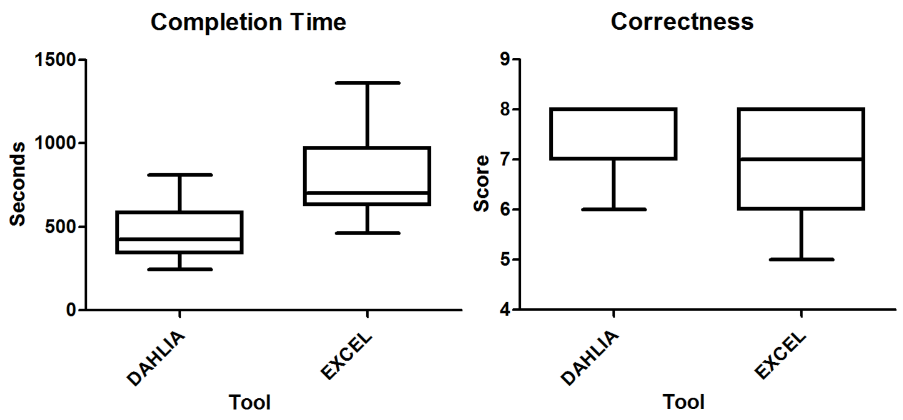


Figure 7.18: Box plots for completion time and correctness.

The greatest difference to observe between the two groups is about the average completion time per task. Except for T3 which is almost similar, DAHLIA 2.0 constantly outperforms Excel. It indicates that DAHLIA 2.0 provides users with faster functionalities to access necessary information.

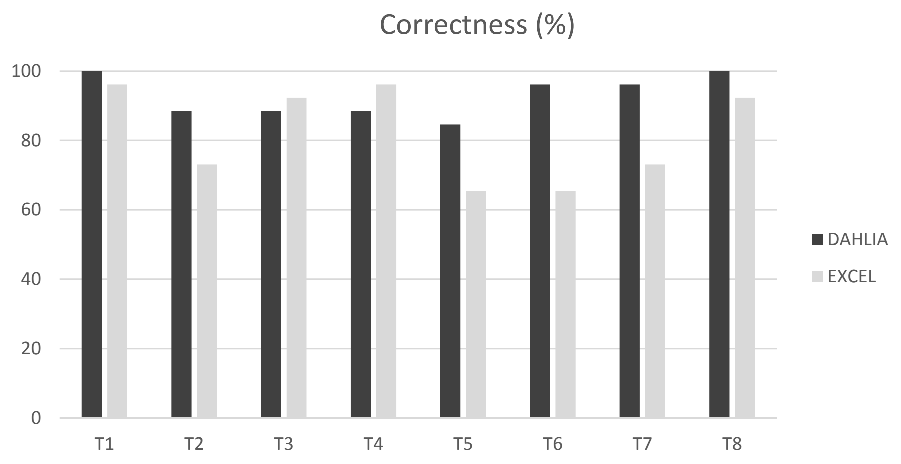
7.4.7 Threat to Validity

Internal Validity

The internal validity is the degree to which the results are attributable to the independent variable and not some other uncontrolled factors.

Participants. For this experiment, we selected students from last year of Bachelor, evening program and Master. Their participation was not mandatory, but voluntary. Since every participant already followed the course of database engineering, we claim that they all have the background enough to understand and answer the questionnaire. However, we cannot consider that all students had enough competencies in the field of program comprehension. Secondly, students' motivation is another important factor to consider; whereas the students were voluntary, their (lack of) motivation before and during the experiment could influence the results.

Baseline. As we looked for a fair baseline on which we could compare DAHLIA 2.0, we rejected Eclipse IDE. Indeed, we consider that the time necessary to manually (1) detect database accesses from the source code and (2) recover the exact executed query value are time-consuming tasks and would confer a considerable (and unfair) advantage to DAHLIA 2.0. Whereas our choice of selecting Excel as baseline could affect the performance of the control group, we are convinced that this baseline was chosen on a fair basis.



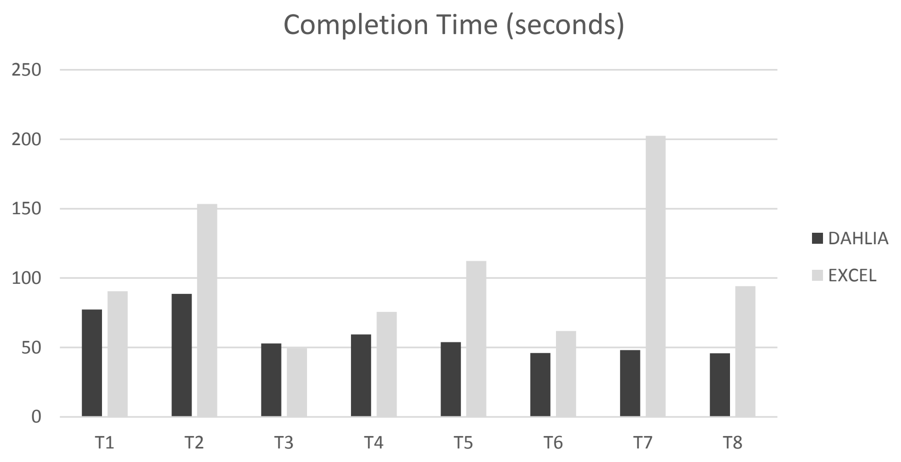


Figure 7.19: Average correctness (on the top) and completion time (on the bottom) per task.

Tasks. The choice of tasks may have been biased to the advantage of DAHLIA 2.0.

Training. We only trained the participants using DAHLIA 2.0 to answer the questionnaire and this may have influenced the experiment's results. However, we presented the Excel sheets and described their structures to the control group before starting the experiment. Moreover, answering the questionnaire with Excel only requires basic knowledge of Excel and therefore, we think that training the control group was not necessary.

External Validity

The external validity is the degree to which the results of the experiment can be generalized.

Used object system. The representativeness of the system we chose to perform our experiment may be considered as a threat. However, this real-life open-source system is designed for a real-world domain application (i.e. health care in developing countries) and has a realistic size and complexity.

Participants. The representativeness of the selected students is a potential threat. Indeed, undergraduate students may not be considered as fully representative population of the target profile of potential users of DAHLIA 2.0.

Tasks. The representativeness of the tasks is also a potential threat; the questions may not reflect real program comprehension tasks — it is possible that other kinds of questions matter more than the ones we considered.

7.5 Concluding Remarks

We presented DAHLIA 2.0, a novel visualization tool that allows us to analyze the database usage of dynamic and heterogeneous systems by visualizing the links between the source code and the database. It aims to support database-program co-evolution in a DISS. Our tool can deal with systems using several database access technologies together like ORM. The DAHLIA visualization relies on a data model detailed in Section 5.2.2 and could become technology-independent with some minor adaptations.

Finally, we presented a controlled experiment aiming to quantitatively evaluate how DAHLIA 2.0 can influence completion time and correctness of performing comprehension tasks pertaining to the program-database communication. The results of the experiment indicate that DAHLIA 2.0 leads to an improvement of completion time and correctness. The obtained results are statically significant, which means that our visualization approach is able to help developers in the context of program-database co-evolution. In particular, we observed that the DAHLIA group (1) spent 41.99% less time to achieve the tasks and (2) reached a higher level (+7.33%) of correctness when answering the questions.

Roadmap

In this chapter, we have presented DAHLIA 2.0, a 3D visualization tool allowing the analysis of the database-program interactions. We also presented a controlled experiment for the empirical evaluation of DAHLIA 2.0 and observed that our visualization approach is able to help developers in the context of program-database co-evolution.

The next chapter (Chapter 8) presents three direct applications of the approaches presented in the previous chapters to other fields.

CHAPTER 8

OTHER APPLICATIONS

This chapter[a] presents three direct applications of the approaches presented in the previous chapters, namely (1) concept location for DISS, (2) database reverse engineering and (3) database schema evolution in schema-less NoSQL data stores.

[a]This chapter extends three papers. The first one [Nagy et al., 2015] was published in the proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015). The second paper [Meurice et al., 2014] was published in the proceedings of the 30th European Conference in Software Maintenance and Evolution (ICSME 2014). The third paper [Meurice and Cleve, 2017] was published in the proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017).

8.1 Introduction

This Chapter illustrates the benefits of different approaches presented in the previous chapters. We show the benefits of our approaches by introducing three direct applications. The first application is static analysis solution for a typical concept location problem for data-intensive systems (see Section 8.2); the second application is a database reverse engineering process to recover implicit referential integrity constraints in legacy systems (see Section 8.3); the third application is an automatic approach supporting schema evolution in schema-less NoSQL data stores (see Section 8.4).

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

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

As illustration, there are several forum questions and blog posts like "*Finding the origin of a Query*"[1] and "*Backtrace from SQL query to application code*"[2], which address this problem of identifying the exact code location where a query is executed. Typical scenarios for this task are when queries need to be optimized for performance, or when they cause failures (e.g., a syntactic error or a deadlock issue). But, because of the lack of static tools, a dynamic solution is almost always recommended. Indeed, dynamic techniques exist to trace the query on the database or client side. However, dynamic analysis cannot help us in certain situations. Suppose that the user of the application experiences performance issues at the database; she/he identifies the query which causes the performance drop back in the log files of the database and sends a bug report. Since the problem occurred in the database and was reported by it (the client was not directly affected), we do not have a stack trace in the bug report. How can we determine where the query was prepared in the source code? We must reproduce everything exactly as the user did, which might prove impossible if we depend on the (possibly confidential) data stored in the database. In such situations, a static approach seems more appropriate.

Therefore, we adapted our static analysis solution presented in Chapter 4 for addressing a typical concept location problem for data-intensive systems: "*where was this query executed?*". Specifically, we adapted our static technique for identifying the exact source code location from where a given SQL query was sent to the database server. This task can become really hard as the complexity of a system grows, especially in the case of languages where queries are constructed dynamically. For example, simple grep or code search techniques are not sufficient for systems where thousands of queries are constructed via string operations and methods deep in the call hierarchy. Moreover, persistence frameworks (like Hibernate and JPA) can hide the query construction from developers, further complicating the debugging of such issues.

8.2.1 Approach

Figure 8.1 shows the main steps of our concept location approach. The process starts with the developer who specifies the SQL query that she/he would like to find in the source code of an application. Then we analyze the source files including the

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

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

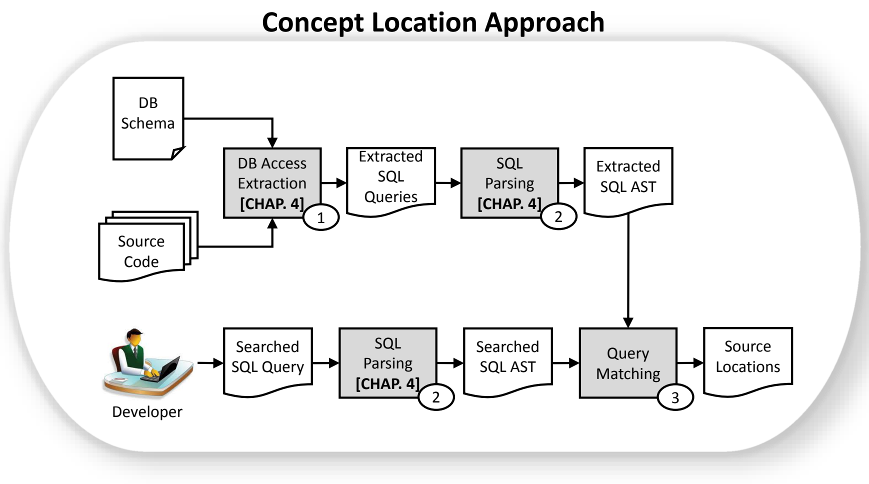


Figure 8.1: Overview of the approach, where the main steps are in numbered boxes with their respective inputs and outputs.

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

Listing 8.1: Example of HQL query construction.

database schema by executing our approach presented in Section 4.2. The result is a set of matching queries and the locations of the method invocations that send them to the database.

Database Access Extraction. This step aims at analyzing the application source code for detecting and extracting SQL queries sent to the database server via JDBC, Hibernate and JPA. This step is ensured by the JDBC/Hibernate/JPA analyses of our static analysis approach, presented in Section 4.2. This step results in a set of database accesses. Each Hibernate and JPA access is translated in its corresponding SQL form.

Listing 8.1 shows a typical method that constructs a HQL query to list some books in a database, and Listing 8.2 shows the query string that we can extract from this method. Listing 8.3 shows the corresponding SQL form obtained after translation.

SQL Parsing. Once we have all the potential data access points and all the native or translated SQL queries, the next step is to compare all these with the query that the developer is interested in. We perform this comparison at the level of Abstract Syntax Trees (ASTs) in order to have more flexibility for the comparison and to be able to

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

Listing 8.2: HQL query extracted from the code sample in Listing 8.1.

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

Listing 8.3: A SQL query translated from the HQL query in Listing 8.2.

handle unresolved query fragments. We reuse the SQL parsing process detailed in Section 4.2.2, which is able to parse statements with unresolved query fragments. In Figure 8.2, we can see an AST constructed from the statement given in Listing 8.3.

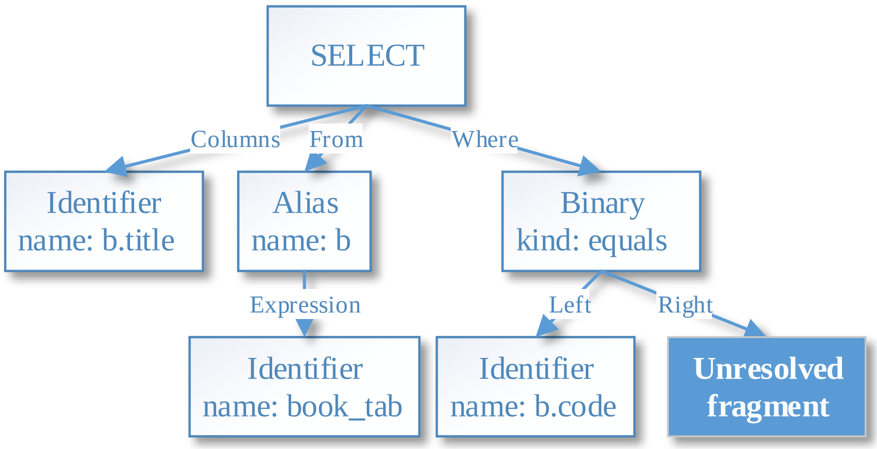


Figure 8.2: Illustration of the AST of the query in Figure 8.3

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

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

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

Figure 8.3 shows a sample AST which is in *match* relation with the AST in Figure 8.2. All the nodes are exactly the same, except the `Literal` node, which matches the 'unresolved fragment' node in the tree.

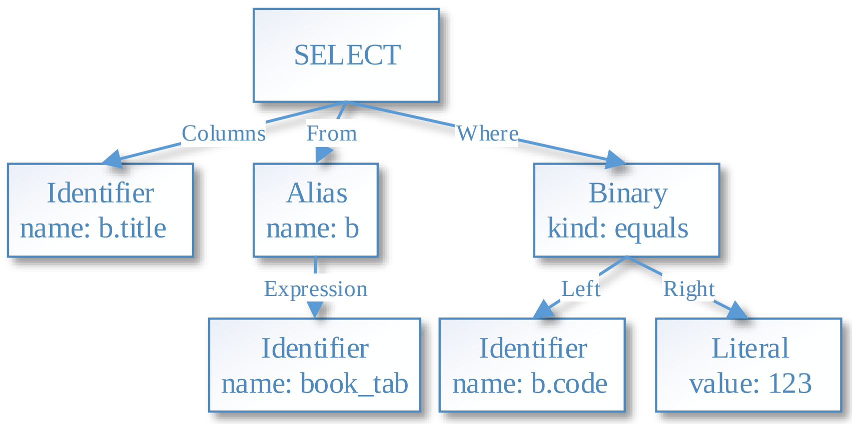


Figure 8.3: Example AST matching the AST in Figure 8.2

8.2.2 Evaluation

We implemented our approach for systems written in Java and accessing a database via JDBC/Hibernate/JPA. To evaluate it, we tested the implementation on OSCAR and OpenMRS. We claim that the size and the architecture of these systems allow us to demonstrate the efficiency of our approach in a real-world environment.

Query Extraction and SQL Parsing. Table 8.1 shows the number of SQL queries that we successfully extracted and parsed from the source code of OSCAR and OpenMRS. Both systems follow the Data Access Object (DAO) pattern (but not strictly), hence most of the queries are prepared and sent to the database from DAO classes. It can be seen, however, that OSCAR mixes the usage of JDBC, Hibernate and JPA, while OpenMRS uses Hibernate more extensively. The column of successfully parsed statements shows the number of queries for which we could successfully construct an AST. Queries that we cannot parse might contain syntactic errors or language constructs that our parser cannot handle in its current implementation state.

To play the role of the developer who seeks a problematic query, we collected SQL queries from execution traces of usage and testing scenarios. Both OSCAR and OpenMRS have a test database available in their source repository, and their developers intensively use unit tests for basic functionalities and DAO implementations too. OSCAR and OpenMRS have 1,311 and 3,258 test cases respectively, in their unit testing framework. We executed all these test cases and used log4jdbc to trace database usage. For all the collected SQL queries, we saved the actual stack trace too.

System	LOC	Tables	JDBC Queries		HQL/JPQL Queries	
			Extracted	Parsed	Extracted	Parsed
OSCAR	2 054 940	480	123 661	123 298	32 456	9 005
OpenMRS	301 232	88	77	73	687	151

Table 8.1: Size metrics of the systems and the number of queries extracted and successfully parsed

Then we filtered queries (based on their traces) that were sent to the database via JDBC or Hibernate HQL/JPQL and we tried to locate them with our method.

System	Test Scenario	Q	TP	TPR	FP	FPR	Max	Avg	Var
OSCAR	Billing	103	102	0.99	0	0.00	19	9.97	8.49
OSCAR	Change Password	3	2	0.66	0	0.00	2	1.50	0.50
OSCAR	First Login	3	2	0.66	0	0.00	2	1.50	0.50
OSCAR	New Demographic	3	2	0.66	0	0.00	2	1.50	0.50
OSCAR	Request Consultation	101	100	0.99	0	0.00	19	1.50	10.14
OSCAR	Send Message	5	3	0.60	0	0.00	2	1.33	0.44
OSCAR	Update User	3	2	0.66	0	0.00	2	1.50	0.50
OSCAR	Writing Prescriptions	3	2	0.66	0	0.00	2	1.50	0.50
OSCAR	Unit Tests	1005	650	0.65	14	0.01	4	1.51	0.50
OpenMRS	Unit Tests	39	34	0.87	0	0.00	4	2.11	0.49

Table 8.2: True and false positive ratio of locations reported for JDBC queries

System	Test Scenario	Queries	TP	TPR	FP	FPR	Max	Avg	Var
OSCAR	Add Provider	20	19	0.95	0	0.00	1	1.00	0.00
OSCAR	Add Role	22	20	0.91	0	0.00	1	1.00	0.00
OSCAR	Billing	702	294	0.41	2	0.01	5	1.08	0.15
OSCAR	Change Password	39	33	0.84	0	0.00	1	1.00	0.00
OSCAR	First Login	77	60	0.78	0	0.00	1	1.00	0.00
OSCAR	New Demographic	67	47	0.70	1	0.01	5	1.12	0.23
OSCAR	Request Consultation	498	150	0.30	2	0.01	5	1.06	0.12
OSCAR	Send Message	43	36	0.84	1	0.02	2	1.02	0.05
OSCAR	Update User	56	39	0.69	0	0.00	1	1.00	0.00
OSCAR	Writing Prescriptions	100	67	0.67	1	0.01	2	1.03	0.05
OSCAR	Unit Tests	1559	950	0.61	23	0.01	5	1.10	0.19
OSCAR	Unit Tests	317	268	0.84	0	0.00	4	1.06	0.13

Table 8.3: True and false positive ratio of locations reported for Hibernate queries

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

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

Listing 8.4: An example SQL query from the Billing scenario of OSCAR.

```
public Object[] getUnitPrice(String bcode, Date date) {
  String hql = "select bs from BillingService bs where bs.serviceCode = ? and bs.
       billingserviceDate = ?";
  Query query = entityManager.createQuery(hql);
  query.setParameter(1,bcode);
  query.setParameter(2, getLatestServiceDate(date,bcode));

  List<BillingService> results = query.getResultList();
  ...
}
```

Listing 8.5: The original HQL query and the Java code which prepares the query in Listing 8.4 (`BillingServiceDao.java`).

Table 8.2 and 8.3 show the number of queries and the true positive (TP) or false positive (FP) location sets, respectively, with the true or false positive ratios (TPR, FPR). Queries where the set of locations reported is empty are false negatives ($Queries - TP - FP$). Max, Avg, $Var\ Match$ stands for the maximum, average and variance values for the sizes of the sets of locations reported.

The results of JDBC reveal that we were able to identify most of the queries of OSCAR usage scenarios with a TPR (which is actually equal to the recall in our case) of 60-99%.

The results of Hibernate look promising too. Except for two scenarios, we were able to identify the origin of 60-95% of the queries by reporting almost everywhere just the matching location (see Avg. Match values). These results could probably be improved by getting a better parsed-extracted ratio for HQL/JPQL queries in OSCAR (see Table 8.1).

Listing 8.4 shows a sample SQL query that we traced from the Billing scenario of OSCAR and its origin, which is shown in Listing 8.5.

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

8.2.3 Discussion

Observations. As previously discussed, there are several forum questions and blog posts which try to address this problem of identifying the origin of an executed query, and because of the lack of static tools, they almost always recommend a dynamic solution. In contrast, as we pointed out earlier, dynamic analysis is not always feasible. Here, our goal was to devise a static analysis approach and to demonstrate its potential use. Preliminary results show that a static technique can achieve good precision and recall with good true/false positive ratio in locating SQL queries sent to the database over JDBC, Hibernate or JPA.

Limitations. Our implementation is limited to some technologies. The query extraction technique is limited to JDBC, Hibernate and JPA, and able to extract SQL, HQL and JPQL queries. Hence, we had to make the assumption for our evaluation that a developer seeks these kinds of queries and when we collected SQL strings, we filtered and kept only JDBC/Native/HQL/JPQL queries.

8.3 Establishing Referential Integrity in Legacy Information Systems

Most modern relational DBMS have the ability to monitor and enforce referential integrity constraints (RICs), i.e., foreign key (FK) constraints. In contrast to new applications, however, heavily evolved legacy information systems may not make use of this important feature, if their design predates its availability. Such applications must be reengineered in order to benefit from automated integrity enforcement. Although the database layer is not the only option for enforcing RICs (presentation and application layers are other options), pushing enforcement in this layer is often seen as the most reliable option to control many concurrent ”channels” entering data into the system.

From a high level perspective, this reengineering process consists of two steps, namely FK identification and FK implementation. Research activity in this area has primarily focused on the first step (identification), which can be viewed as a form of design recovery. A wealth of different methods and tools have been proposed to recover FKs from a variety of data sources, including the database schema, application code, data instances, and documentation. While many FK identification methods have been proposed, empirical evidence about their comparative effectiveness in real-world industrial settings remains rare.

In contrast to many other research works that start by proposing a new or improved solution to the above described reengineering problem, followed by a validation with problem case studies (often hand picked to make a point), we address the actual problem in the context of OSCAR, a real-world, large-scale legacy system in the healthcare industry. As a result of our analysis, we find that many of the assumptions commonly made in database reengineering methods and tools do not readily apply in practice. Based on our problem analysis we devise a process for reengineering legacy information systems with respect to establishing referential integrity constraints, incorporating, combining and extending existing reengineering methods.

We report on empirical results of implementing this process in the context of our problem case study system. Our results suggest that the process of reengineering legacy information systems with respect to establishing referential integrity constraints may be considerably more complex than is commonly assumed. It must be understood as an incremental detection process.

8.3.1 Problem Case Study: OSCAR

As we mentioned it in Section 3.5, the database schema of the OSCAR distribution does not contain any information on relationships between tables (foreign keys) and no documentation is available about the schema. We learned that the missing relationships were due to the evolution history of OSCAR. OSCAR has been using MySQL as its DBMS platform. MySQL supports the choice of different alternative storage engines. During the first five years of OSCAR development, MySQL did not support a storage engine capable of enforcing referential integrity. Consequently, OSCAR's database implementation does not make significant use of FK constraints but rather consists of seemingly unrelated tables. Over the last several years, OSCAR has been migrating to MySQL's newer InnoDB storage engine, which provides full support for referential integrity enforcements. Since then, more recently developed parts of the system have made use of FK constraints. Still, the vast majority of the database tables remain without any explicit relationships in the schema. This situation has been a frequent source of frustration in the OSCAR developer community as it impedes program understanding and maintenance. It has also raised concerns with respect to the integrity of patient health information and, ultimately, patient safety. Therefore, it has been a goal to reengineer OSCAR with respect to establishing more referential integrity constraints.
We encountered a number of challenges in our case study.

Size. One obstacle in this process is the sheer size of the database schema. With close to five hundred tables and some of the larger tables comprising over thousands of columns, identifying FKs cannot be a manual process but requires automated tool support.

Multi-paradigm architecture. As discussed in Section 3.5, another challenge is the unevenly evolved nature of the OSCAR architecture, which uses a multitude of different paradigms to access the database. Some older application modules still use embedded (dynamic) SQL queries, while newer modules use object-relational middleware descriptors (Hibernate mapping files), and yet newer application code uses code annotation tags based on the JPA standard. Therefore, no single method for detecting FKs in application code is likely to recall all relevant relationships.

Confidential data. Knowledge about the actual database instances is an important prerequisite for the process of identifying RICs. It is not uncommon that the data in

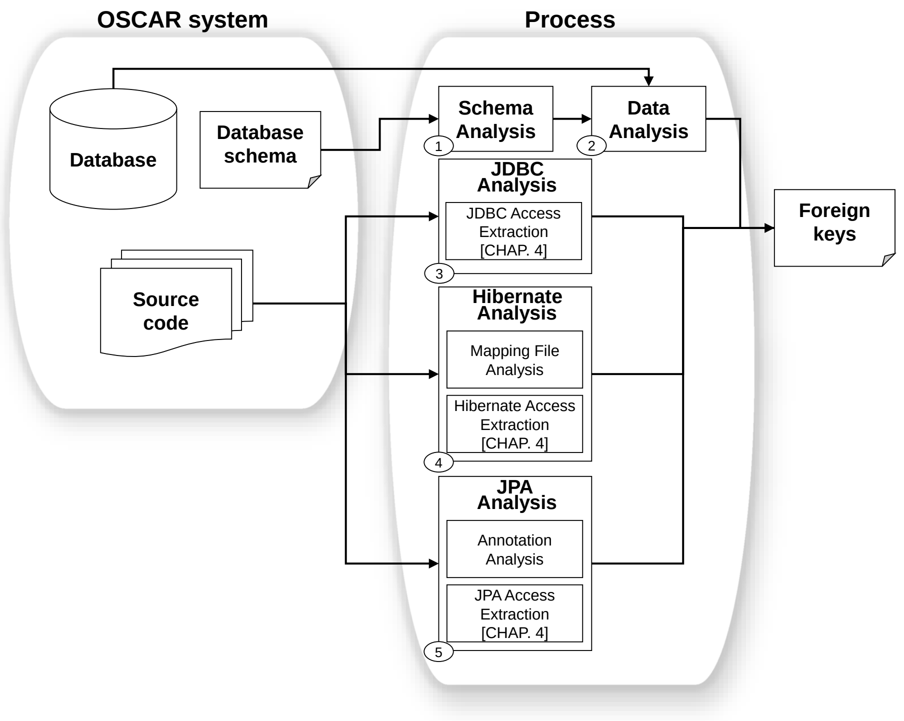


Figure 8.4: Overview of our FK detection process.

legacy information systems is considered business confidential. However, patient records are among the most sensitive and highly regulated information items in any industry and they cannot commonly be made available for the purpose of software engineering, even under non-disclosure agreements. We had to create software and a process to securely encrypt the data prior to FK analysis and attain approval from the University ethics board prior to our reengineering study.

No oracle available. Since (1) the vast majority of the database tables remain without any explicit FK constraints in the schema and (2) no previous reengineering process has been done yet on the OSCAR system, we do not have any oracle to systematically evaluate the proposed approach. Instead, involving the OSCAR developers in the establishment of of a ground truth may be required.

8.3.2 Foreign Key Detection Process

The reengineering process, depicted in Figure 8.4, applied in this study starts with the identification of implicit integrity constraints through the **triangulation** of several RIC identification techniques. We present a process to address the RIC detection in a legacy system through the joint analysis of multiple sources of information: the database schema, the database contents and the program source code. The results obtained by each analysis technique are then combined in order to find a certain number of **likely** foreign key candidates. In the following, we describe each analysis step we follow in our reengineering process.

Schema Analysis. The Schema Analysis process is guided by the primary key constraints found in the tables of the schema. Each column $colPK$ contained in a primary key of a table is used to search for other columns in the database schema that could reference it. Algorithm 11 specifies this process. We use *tabPK* and *colPK* variables to refer to the table and the column, respectively, of the primary key side.

```
result ← ∅ ;
for tabPK ∈ schema do
    for colPK ∈ tabPK.constraintPK do
        result ← result + SearchForFK(tabPK, colPK);
    end
end
return result;
```

Algorithm 11: *Schema Analysis* algorithm.

In the **SearchForFK** function, columns are searched based on their names and data types (SQL type, length and precision) as we show in Algorithm 12. Variables *table* and *column* are used to refer to the table and the column, respectively, analyzed as a candidate foreign key.

```
procedure SearchForFK(tabPK, colPK)
    result ← ∅ ;
    for table ∈ schema do
        for column ∈ table do
            if column ≠ colPK then
                if column.type = colPK.type & column.length ≥
                  colPK.length & column.precision ≥ colPK.precision &
                  EqualsNames(table, column, tabPK, colPK) then
                    result ← result + (tabPK, colPK, table, column);
                end
            end
        end
    end
    return result;
```

Algorithm 12: *SearchForFK* function.

The **EqualsNames** function returns *true* if the names of the columns and tables analyzed are **compatible**, and returns *false* otherwise. The meaning of **compatible** is based on the partial matching of the table and column names, according to their length. The function checks the length of the column name $colFK$ considered as foreign key candidate. If the length ≥ 5 characters we check whether the target table name $tabPK$ and/or the target column name $colPK$ is included in $colFK$ (only if its length is > 2, otherwise we consider that this name is not meaningful enough). If the length of $colFK$ is < 5 characters we do not check if $tabPK$ is contained in $colFK$ (because we consider that names are not meaningful) and we only check the length of $colPK$. If it is > 2, we check if $colPK$ is contained in $colFK$. Otherwise, we could suppose that $colPK$ has a name like **'id'** or something similar. In this scenario, a specific check is performed: we eliminate in $colFK$ the occurrences of $colPK$ and some other special characters like **'_'**. Then, we verify if the resulting name is part of $tabPK$[3].

Let us illustrate this using an example. We could have a $colFK$ named **'prid'** which would be analyzed in relation to a $colPK$ named **'id'** in a table named **'provider'**. After elimination of the $colPK$ from $colFK$, we would have the string **'pr'** which would be contained in $tabPK$.

Data Analysis. The Data Analysis process utilizes the results generated by the **Schema Analysis** process as starting point. This approach is usually a necessity for large-scale legacy databases, as a brute-force data analysis with respect to detecting all potential foreign keys is usually computationally prohibitive.

```
procedure SearchForFK(tabPK, colPK)
    result ← $\emptyset$ ;
    for (tabPK.colPK, tabFK.colFK) $\in$ set(tabPK.colPK, tabFK.colFK) do
        countFKReg ← select count(*) from tabFK;
        matching ← (select colFK from tabFK) $\cap$ (select colPK from tabPK);
        percentage ← (macthing $\times$ 100)/countFKReg;
        if percentage $\geq$ threshold then
            result ← result + (tabPK, colPK, tabFK, colFK);
        end
    end
    return result;
```

Algorithm 13: *Data Analysis* algorithm.

Algorithm 13 shows how the data analysis is applied. Taking a set of foreign key candidates, the algorithm calculates the matching of values involved on each candidate. This matching defines how many values in *tabFK.colFK* can be found in *tabPK.colPK*. This matching value must be measured in relation to the number of rows in *tabFK* to calculate the percentage of matching values. The number of rows in *tabPK* is reported for better interpreting that percentage.The algorithm is set up by means of a **threshold** value which is established to only return candidate foreign keys *(tabPK, colPK, tabFK, colFK)* having a percentage value above the threshold value.

JDBC Analysis. Our JDBC Analaysis process enables to analyze the programs source code in order to identify, parse and exploit the SQL queries using the JDBC API. This

[3]Those numeric bounds were arbitrarily chosen based on our experience; however, we are aware that a further study might be needed to calibrate those values with precision.

```
<class name="TicklerUpdate" table="tickler_update">
<many-to-one name="tickler" class="Tickler"
column="tickler_no" update="false" insert="false" lazy="false" />
```

Listing 8.6: Example of many-to-one relationship defined in Hibernate mapping file.

process utilizes our static analysis approach presented in Chapter 4, in particular the JDBC access extraction process (described in Section 4.2.2), in order to detect and extract SQL queries. Once these queries have been retrieved, they are parsed to extract the FROM and WHERE clauses. Analyzing those clauses permits to detect **join** conditions and to infer candidate foreign keys. The join condition is usually expressed by using equality conditions between columns (e.g., **WHERE columnA = columnB**), but other kinds of "join" can be expressed by using nested queries (e.g., **WHERE columnA in (select columnB FROM ...)**). We thus analyze the content of the FROM and WHERE clauses, searching for a join condition and extracting the tables and columns implied in candidate foreign keys.

Hibernate Analysis. A large part of the OSCAR applications uses the Hibernate ORM to access the database. Hibernate allows developers to map Java classes to database tables. Those mappings are usually declared in a mapping file (an XML document) that instructs Hibernate how to map the Java classes to the database tables. We consider the Hibernate XML mapping files as another possible way to infer implicit foreign keys. Although an ORM such as Hibernate offers an abstraction layer permitting to ignore the underlying database structures (and thus the presence of FKs), we consider that some legacy systems could use Hibernate to access legacy databases with missing constraints (like OSCAR). Our Hibernate mapping file analysis searches in each mapping file for a **’class’** tag, where an entity name is mapped to a table name by means of **’name’** and **’table’** attributes, respectively. If both names are equal, **’table’** attribute could be omitted. In a similar way, the attributes in an entity are declared by a **’property’** tag and **’name’** and **’column’** attributes. Declarations of RICs can be defined using the following tags: **’one-to-one’**, **’many-to-one’**, **’one-to-many’** and **’many-to-many’**. Different kinds of RICs are permitted in a mapping file. We illustrate below some of the most common techniques:

- *Many-to-one relationships.* In Listing 8.6, the developer defined a many-to-one constraint between the *tickler* table and the *tickler_update* table mapped to *TicklerUpdate* class. In such a case, we can infer a foreign key from table *tickler_update* to table *tickler*. Our Hibernate parser identifies the **’name’** attribute as the foreign key column, and the **’class’** and **’column’** attributes as the target primary key.
- *One-To-many relationships.* Inversely, in Listing 8.7, the developer defined a one-to-many constraint between the *tickler_update* table and the *tickler* table mapped to the *Tickler* class. Here again, we can infer the foreign key between both tables.

```
<class name="Tickler" table="tickler">
<id name="id" type="integer" column="tickler_no">
  <generator class="native" />
</id>
<set name="updates" inverse="true" cascade="save-update" sort="natural" lazy="false">
  <key column="tickler_no" />
  <one-to-many class="TicklerUpdate" />
</set>
```

Listing 8.7: Example of one-to-many relationship defined in Hibernate mapping file.

```
<class name="Site" table="site">
<set name="providers" table="providersite" lazy="true" inverse="true">
  <key column="site_id" />
  <many-to-many column="provider_no" class="Provider" /> </set>
```

Listing 8.8: Example of many-to-many relationship defined in Hibernate mapping file.

```
<class name="Tickler" table="tickler">
<property name="numUpdates"
  formula= "(select count(*)
    from TicklerUpdate tickler
    where tickler.tickler_no=tickler_no)"/>
```

Listing 8.9: Example of SQL query declared in Hibernate mapping file.

- *Many-to-many relationships*. In Listing 8.8, the developer defined a multi-valued association. The Hibernate parser identifies the **'name'** attribute of **'set'** tag as a foreign key column, and the **'class'** and **'column'** attributes contained in **'many-to-many'** tag as the target primary key. But in this case, since an intermediate table must be referenced for both foreign keys, the **'table'** attribute in the **'set'** tag is needed to refer to an intermediate table name.
- *SQL query declarations*. Hibernate also allows developers to use SQL queries directly in the mapping file. Those queries could be a good indicator for inferring RICs too, especially when the query consists of a join between two tables. Listing 8.9 shows an example of SQL query defined in a Hibernate mapping file.

Furthermore, the Hibernate analysis also utilizes our static analysis approach presented in Chapter 4, in particular the Hibernate access extraction process (described in Section 4.2.3) in order to detect and extract HQL queries from the source code. Once extracted, they are translated in their corresponding SQL form and analyzed for detecting join conditions and candidate foreign keys.

JPA Analysis. Unlike Hibernate, JPA uses Java code annotations rather than XML mapping files to specify how persistent objects and their relationships are mapped to relational table structures. The most recent OSCAR components use JPA annotations rather than Hibernate mapping files. The annotation analysis process detects and

```java
@Table(name = "tickler")
public class Tickler {  ...
  @OneToMany(fetch=FetchType.EAGER)
  @JoinColumn(name="tickler_no", referencedColumnName="tickler_no")
  private Set<TicklerUpdate> updates = new HashSet<TicklerUpdate>();
  ... }
```

Listing 8.10: Example of one-to-many relationship defined with JPA annotation.

analyzes JPA annotations within the source code, in order to recover referential relationships. For each JPA entity file a **’Table’** annotation is searched, where an entity name is mapped to a table name by means of the **’name’** attribute. If both names are equal, the **’Table’** annotation can be omitted. Declarations of RICs are defined using one of the following annotations: **’ManyToOne’**, **’OneToMany’** and **’ManyToMany’**. For instance, Listing 8.10 shows an example of **one-to-many** JPA annotation which expresses the same RIC as in our Hibernate example. The **’JoinColumn’** annotation contains attributes to define the foreign key column (**’name’**) and the column of the target primary key (**’referencedColumnName’**). The table name containing the RIC is obtained from the entity class, and the table name referenced by the RIC is obtained from the class type in the attribute defining the relationship.

Furthermore, the JPA analysis also utilizes our static analysis approach presented in Chapter 4, in particular the JPA access extraction process (described in Section 4.2.4) in order to detect and extract JPQL queries from the source code. Once extracted and translated in their corresponding SQL form, those queries are analyzed to exploit the join conditions and to infer candidate foreign keys.

8.3.3 Results

As described above, we have implemented 5 different techniques for recovering implicit FKs. After applying those techniques on the OSCAR system, we extracted 1,899 FK candidates. Figure 8.5 illustrates the distribution through the 5 techniques: 1,818 FK candidates were detected by the *schema analysis*; 291 by the *data analysis*; 28 by the *Hibernate analysis*; 32 by the *JPA analysis*, and 50 by the *JDBC analysis*.

Another iteration was required for further exploiting those first results. We defined a list of criteria allowing us to accept a FK candidate. Each candidate FK respecting at least one of those criteria is accepted:

(a) The FK is proposed by the *schema analysis* and has a *matching percentage* above or equal to 90%.

(b) The FK is proposed by the *Hibernate analysis*.

(c) The FK is proposed by the *JPA analysis*.

(d) The FK is proposed by the *JDBC analysis* and it refers to a *primary/unique key*.

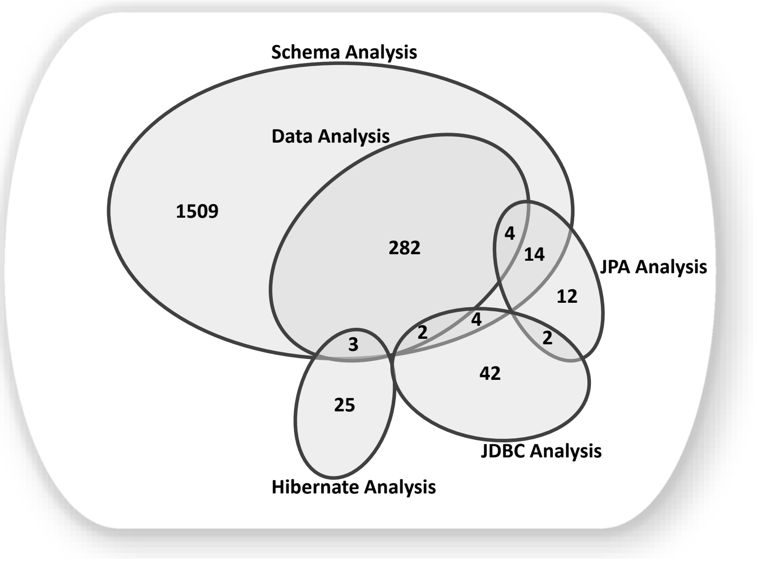


Figure 8.5: Initial report

After applying those criteria, we moved from 1,899 potential to 215 accepted candidates. The 1,684 remaining ones are considered as *unlikely*. In order to reach higher precision in our results, we also considered 4 rejection criteria:

(a) *Matching*. The *unlikely* candidates having a data *matching value* lower than 90% are rejected.

(b) *Bi-directionality*. The *unlikely* candidates such as there exists an *accepted* candidate in the opposite direction, are rejected. An encountered case of rejected candidate, due to the bi-directionality property, is illustrated by Figure 8.7.

(c) *Unicity*. The *unlikely* candidates such as there exists an accepted candidate defined on the same column(s), are rejected. An encountered example of rejected candidate, due to the unicity property, is depicted in Figure 8.8.

(d) *Transitivity*. The *accepted* candidates that could be transitively derived from other accepted candidates are rejected. A case of rejected foreign key, due to the transitivity property, is given in Figure 8.9.

Figure 8.6 represents the distribution of the *accepted* candidates through the 5 information sources after having considered our rejection criteria: 146 *unlikely* candidates have been *rejected* because they do not respect the minimal matching value; 1,219 *unlikely* candidates have been *rejected* by unicity; 23 *unlikely* candidates by bi-directionality while 37 *previously-accepted* candidates have been *rejected* by transitivity.

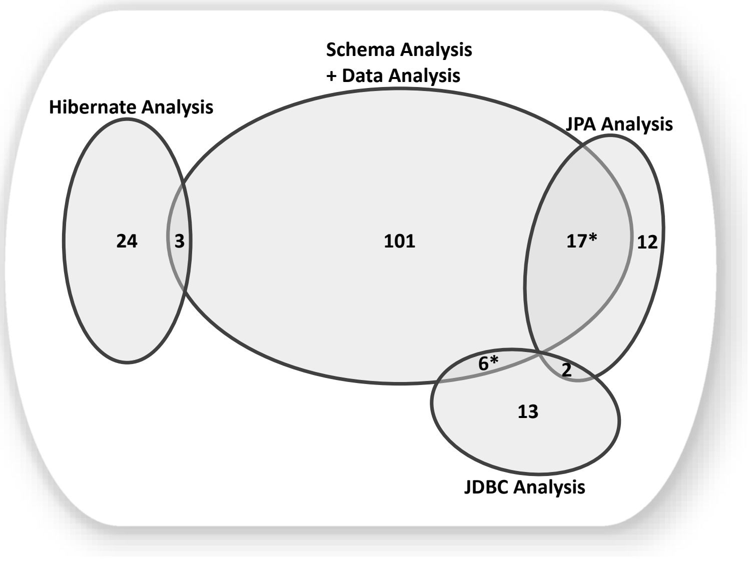


Figure 8.6: Accepted FKs using our accept./reject. criteria.

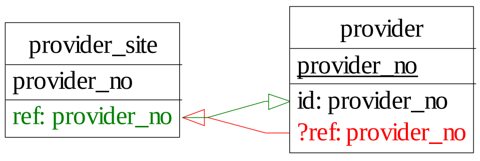


Figure 8.7: Example of candidate rejected due to the bi-directionality property.
`provider[provider_no]` → `provider_site[provider_no]`
is rejected.

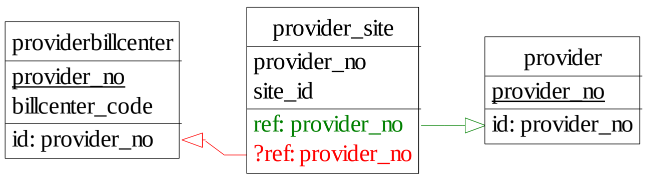


Figure 8.8: Example of candidate rejected due to the unicity property.
`provider_site[provider_no]` → `providerbillcenter[provider_no]`
is rejected.

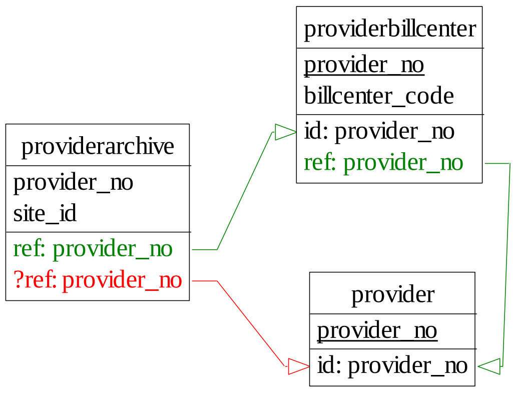


Figure 8.9: Example of candidate rejected due to the transitivity property. `providerarchive_site[provider_no] → provider[provider_no]` is rejected.

8.3.4 Discussion

Observations. The results obtained when identifying FK candidates in OSCAR yield some interesting concluding observations. First, we observe that the different sources of information have different levels of reliability. Although we do not know the actual list of implicit foreign keys that are valid in OSCAR, we can already say that the schema analysis technique may lead to overly noisy results when used in isolation. However, there is no perfect source of information that would, alone, be sufficient for identifying all implicit FKs. For instance, while the Hibernate mapping file and the JPA annotations are reliable sources of information, they allowed us to recover a very limited subset of the implicit FKs in the OSCAR schema, i.e., those involved in the most recent tables. This observation directly relates to the evolution history of the system. We saw that the management of implicit RICs in a system may be largely inconsistent over time. Some old RICs have never been explicitly declared, some more recent ones have been specified through Hibernate, and some others have been declared via JPA annotations. Hence, the FK detection approach we propose, based on the **triangulation** of several techniques for confirming/rejecting FK candidates, seems very promising in the context of a legacy system that has been subject to a long evolution history.

Furthermore, the results obtained allow one to analyze, for each FK candidate, the impact of making the FK explicit in the database schema. In case the FK candidate is violated by the data: the impact is at least twofold: (1) the inconsistent data will need to be corrected or discarded and (2) source code modifications will be required in order to ensure the adequate management of the FK constraint everywhere the programs insert, update and delete rows in the related tables. In contrast, when the FK candidate is systematically encoded via Hibernate or through JPA annotations, the impact of its explicit declaration at the database schema side is much more limited.

Limitations. Although it has shown some merits, our multi-source FK identification process suffers from several limitations. First, the threshold of 90% for the data consistency heuristics could be further validated and calibrated with respect to the number of rows in the referencing table. In addition, since we did not have access to a ground truth, it was difficult for us to precisely quantify the reliability and the complementarity of the different identification techniques we combine. This will be a prerequisite to further improve our triangulation process and devise a more accurate FK candidate ranking method. Finally, as we mentioned above, some OSCAR tables involved in a FK candidate being empty, our results are partially incomplete as well.

Future work. We anticipate several directions for future work in the context of FK detection. First, we intend to further investigate the OSCAR case study, and to involve the developers in the establishment of a ground truth, even partial. Second, we plan to consider other sources of information for the identification and ranking of FK candidates. We think, in particular, of integrating historical information. For instance, let us assume that the history analysis reveals that the same developer has created both tables involved in a FK candidate, this could be seen as an additional confirmation argument. In contrast, if a FK candidate involves two very recently created tables the names of which do not appear in the Hibernate file nor in the JPA annotations, this could be considered as a rejection argument. Last but not least, we intend to devise a tool-supported methodology for assisting developers to incrementally implement identified FK candidates in a legacy software system.

8.4 Supporting Schema Evolution in Schema-less NoSQL Data Stores

NoSQL data stores are becoming increasingly popular in the context of big data software development. These data stores were designed to manipulate big volumes of data that are not organized according to the relational model. NoSQL technologies were introduced to address some relational database limitations: simplicity of design, faster query execution and flexibility. Indeed, most NoSQL data stores are schema-less and can thus manage data with ever-changing structures. In a continuously changing environment, database schema evolution becomes an unavoidable activity and therefore, proposing such a flexibility is a precious asset. Schema-less NoSQL data stores do not require developers to specify a global schema, which makes data evolution simpler. For instance, adding new fields to a data structure can be done at any time and instantly.

However, this flexibility may lead to an increasing data structure entropy within the system. When the schema evolves, the outdated entities must be migrated to fit with the new structures. Nevertheless, migrating data may be time-consuming and expensive; especially when a huge amount of data has to be migrated or when the system is contractually linked to a database-as-a-service provider for all data store

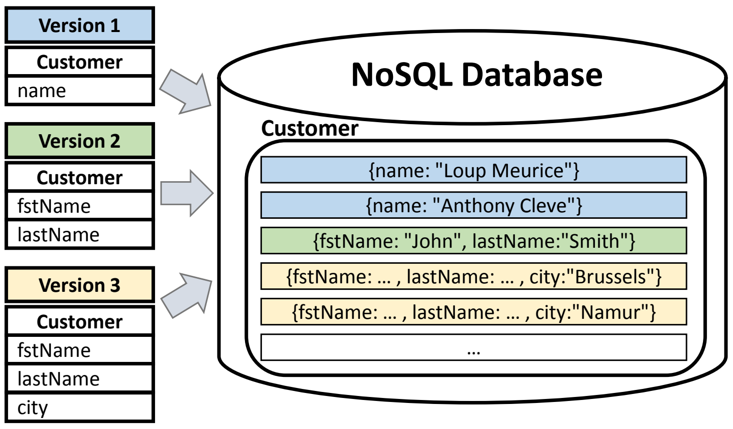


Figure 8.10: Example of the co-existence of legacy and up-to-date entities within the same NoSQL database.

reads and writes. As a consequence, data migration may never be achieved and thus, data entities of different schema versions may co-exist in the data store. Figure 8.10 illustrates an example of co-existence of legacy and up-to-date data entities within the same NoSQL database, after several changes of the data structure. Such an entropy may prove error-prone. For instance, conflictual entities can cause runtime errors and data loss, or can even corrupt the database, if not handled properly. For instance, changing the type of a particular field requires to deal with both the legacy and the up-to-date entities when manipulating data in the program. In other words, in NoSQL data stores, the past belongs to the present, and clearly affects the future. Therefore, understanding schema evolution in schema-less NoSQL databases is essential for future developments.

In *schema-less* data stores, no explicit database schema is declared by developers. Thus, the main source of information concerning the data structures is the source code itself. In particular, the database writes and reads located in the source code give concrete clues about data structures. Among other NoSQL technologies, MongoDB is a schema-less document-oriented database. It stores JSON-like documents in *collections*. Collections are similar to tables in relational databases, and are composed of *fields*[4].
Listing 8.11 depicts an example of Java code manipulating entities from a MongoDB database. Useful information can be extracted from that code sample; the existence of several collections as well as the type of some of their fields can be inferred. Moreover, analyzing how the source code (especially the database-related code) evolved over time can significantly help developers to understand how the schema evolved and thus to prevent potentially severe errors.

In this Section, we presents an automatic historical analysis approach that aims at understanding schema evolution in NoSQL data stores. We present two main novelties: (1) an automatic approach that infers the database schema of a schema-less

[4]https://docs.mongodb.com/manual/

```java
DB db;
public String save(ContributionToSave contributionToSave) {
        BasicDBObject authorQuery = new BasicDBObject("_id", new ObjectId(
             contributionToSave.getAuthor().getId()));
        DBObject author = db.getCollection("author").findOne(authorQuery);
        BasicDBObject showQuery = new BasicDBObject("_id", new ObjectId(
             contributionToSave.getShow().getId()));
        DBObject show = db.getCollection("show").findOne(showQuery);
        addContributionToAuthor(contributionToSave, authorQuery, author, show);
        return "ok";
}

private void addContributionToAuthor(ContributionToSave contributionToSave,
     BasicDBObject authorQuery, DBObject author, DBObject show) {
        BasicDBList contributions = (BasicDBList) author.get("contributions");
        if (contributions == null) {
            contributions = new BasicDBList();
            author.put("contributions", contributions);
        }
        BasicDBObject contribution = new BasicDBObject();
        contribution.put("nick", contributionToSave.getNick());
        BasicDBObject contributionShow = new BasicDBObject();
        contributionShow.put("alias", show.get("alias"));
        contributionShow.put("name", (String) show.get("name"));
        contributionShow.put("ref", new DBRef(db, "show", show.get("_id")));
        contribution.put("show", contributionShow);
        contributions.add(contribution);
        db.getCollection("author").update(authorQuery, author);
}
```

Listing 8.11: Java code example using the MongoDB API to access the database.

NoSQL data store by analyzing the application source code and (2) the application of this approach to the whole system history in order to understand schema evolution and to prevent errors and data losses.

8.4.1 Approach

This Section presents our automatic approach allowing developers to understand and analyze schema evolution in schema-less NoSQL data stores. Our approach, summarized in Figure 8.11, is made up of three phases, namely *schema extraction*, *historical schema extraction* and *exploitation*.

Schema Extraction

As previously explained, useful information about the NoSQL database schema can be extracted by analyzing the source code, especially those code locations accessing the database. The first step of our automatic approach aims to infer the database schema by *statically* analyzing the database accesses from the source code. Our technique is currently implemented for systems using the MongoDB Java driver to communicate with the database. 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 [TIOBE Programming Community Index, 2017]. Moreover, we focus on MongoDB which is currently ranked among the top

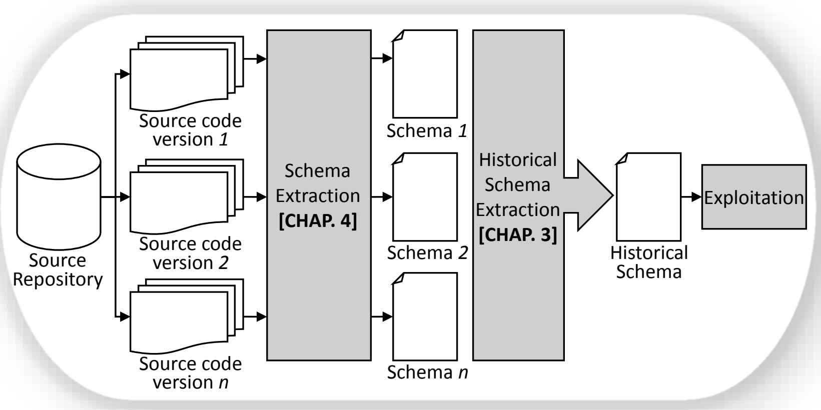


Figure 8.11: Overview of our approach.

five database systems and at first position among the NoSQL database systems [DB-Engines Ranking, 2017].

For describing our technique, we will consider the Java code example depicted in Listing 8.11. If one further observes this code, one can detect the presence of three database accesses, at lines 4, 6 and 25, respectively. (1) Line 4 reads the database to find a particular *author* based on a given identifier; (2) line 6 reads the database to find a particular *show* based on a given identifier; (3) line 25 updates a particular *author*.

Through this example and by inspecting the Java MongoDB API documentation[5], we can make two important observations:

(a) A database access may use one or several selection criteria to create the query (e.g., `authorQuery` at line 4, `showQuery` at line 6 and, `authorQuery` and `author` at line 25). Thus, analyzing the operations performed on those objects *before* the database access execution, brings added information about the collection fields concerned by the selection criteria.

(b) A database access may return a set of objects resulting from the operation (e.g., `author` and `show` variables respectively at line 4 and 6, storing the query results). Thus, analyzing the read operations performed on those objects *after* the access execution, gives indications about the fields they contain.

In other words, information about the database structures may be inferred by analyzing (1) the usage flow of the query inputs **before** query execution and (2) the usage flow of the query outputs **after** query execution. A first algorithm is proposed in Algorithm 14:

Line 1 detects the code locations accessing the database. We achieve this first step by exhaustively listing the set of Java methods provided by the MongoDB API that allows developers to query the database. Once this list established, we use a

[5]https://api.mongodb.com/java/current/

```
foreach access ∈ getAccesses() do
    getCollectionNames(access);
    foreach input ∈ access.inputs do
        analyzeUsageFlowBefore(input);
    end
    foreach output ∈ access.outputs do
        analyzeUsageFlowAfter(output);
    end
end
```

Algorithm 14: Algorithm resolving MongoDB accesses within the source code.

visitor class which parses the Java code and detects any MongoDB database accesses. For each detected access, we have to determine the set of collections that is queried (`getCollectionNames` procedure). This information can be obtained by analyzing instructions in the form `DB.getCollection(String collectionName)`. The answer is in the value affected to `collectionName`. However this value depends on the call graph of the application and the intraprocedural control-flow of the methods. Indeed, building a string value may necessitate to pass through different statements and boolean conditions (e.g., for, while, if-then-else statements). Thus, our static analysis has to consider all possible program paths. Similarly, the string value construction may be done by successive concatenations of string fragments or by using some input parameters of the local method. Therefore, we need to consider the call graph of the application to get the actual values of the string input parameters. To achieve this task of string reconstruction, we use the tool support, dedicated to database access recovery in Java source code, that we developed and presented in Section 4.2. We developed an extended version of this static analysis approach to automatically detect and reconstruct a MongoDB access by exploring the call graph of the application and the intraprocedural control-flow of the methods. Let us come back to our example in Listing 8.11: our static analyzer automatically detects that the database accesses at line 4, 6 and 25 actually query, respectively, the *author*, *show* and *author* collections.

The next step consists in analyzing the input objects which serve as selection criteria for the query creation (`analyzeUsageFlowBefore` procedure). Thanks to the MongoDB API, we pointed out that the creation criteria are mainly expressed by means of the `DBObject`, `BasicDBObject` and `BasicDBList` classes. An instance of those classes is a key-value map that can be stored in the database, where the key represents a field name and the value is the field value. Accordingly, analyzing the usage flow of this map from its creation until the access execution allows us to spot the fields used to define the query. To realize this task, we overloaded our static analyzer so that it can control the usage flow of the inputs. For instance, the database access at line 4 uses a unique selection criterion to create its query, i.e., the `authorQuery` object. By analyzing the usage flow of this given object (and by reusing our string value extractor), our analyzer automatically spots line 3 which actually represents a value assignment to the `author._id` field. It is worth noticing that

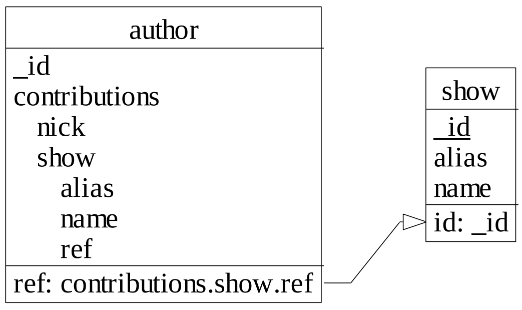


Figure 8.12: Schema automatically inferred by our approach applied to Listing 8.11.

the `analyzeUsageFlowBefore` procedure is actually recursive. Indeed, the value assigned to a particular key in the map may be, itself, an instance of the `DBObject`, `BasicDBObject` and `BasicDBList` classes and thus, a recursive call is needed to analyze this instance.

The final step consists in analyzing the usage flow of the output objects resulting from the database access (`analyzeUsageFlowAfter` procedure). Indeed, analyzing the operations performed on those output objects may reveal new fields of the target collection. This step is similar to the previous one, the only difference being that, instead of analyzing the object usage flow before the database access, it focuses on the object usage flow after the database access. In Listing 8.11, variable `show` contains the result of the database access at line 6. However, analyzing the usage flow of an object after a given event (i.e., a database access) requires the analysis of the application call graph, since the object may be part of the input parameters of a method call. In the example, our static analyzer determines that `show` is used as input in the `addContributionToAuthor` method call (line 7), and visits this method to observe how the object is manipulated. At line 20, 21 and 22, the analyzer detects the read of, respectively, the `alias`, `name` and `_id` fields.

As output, the analyzer returns the database schema fragment which is concerned by each detected database access. Finally, the analyzer merges all the extracted schema fragments in order to obtain a unique schema. Figure 8.12 depicts the schema automatically inferred by our approach when applied to Listing 8.11. Our analyzer is also able to deal with the referential constraints, i.e., *foreign keys*, declared in the source code. At line 22 in Listing 8.11, a referential constraint is declared between `author.contributions.show.ref` and `show._id`. Indeed, `DBRef` allows documents located in multiple collections to be easily linked to documents from a single collection.

The field extraction process also tries to gain information about the types of the fields. For instance, after detecting an access to the `name` field at line 21, our analyzer considers the extracted field as a `String` object.

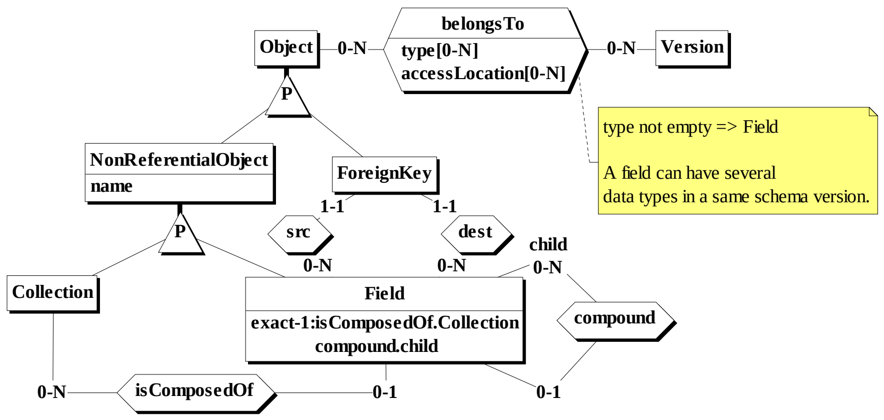


Figure 8.13: Conceptual representation of the historical schema.

Historical Schema Extraction

The second step of our approach aims to apply the schema extraction process to the whole system history by exploiting the versioning system. This step, directly inspired by the historical analysis approach presented in Section 3.2, consists in extracting and comparing the successive versions of the database schema, in order to produce the so-called *historical database schema*. The latter is a representation of the database schema evolution over time, as depicted by the ER model in Figure 8.13. It contains all database schema objects (i.e., `collections`, `fields` and `foreign keys`) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime such as (1) the list of schema `versions` where the object is present and (2) for each version of this list, the code `locations` accessing the object. In addition, each field owns information about its data `type` and its evolution; one can know the data type(s) of any field at any version of the system (particular cases can happen where a same field can have several possible data types in a same schema version). In this way, one has an accurate overview of the field evolution over time which could allow detecting error-prone data type changes.

The historical database schema thus constitutes an integrated representation of the system past and present. Exploiting this historical schema can help to detect potential runtime errors or data corruptions and to facilitate future developments.

Finally, we apply an automatic procedure that colourizes each historical schema object, depending on its age and its liveness. The schema objects depicted in green are still present in the latest schema version. The red schema objects do not belong to the latest version. The colour shade corresponds to the age of the objects. A dark red schema object is an object that has disappeared a long time ago. A light red object is an object that has recently disappeared from the schema. An object depicted in green corresponds to an object that is still present in the latest schema version. The darker the green, the older the object is, and vice versa. An example of

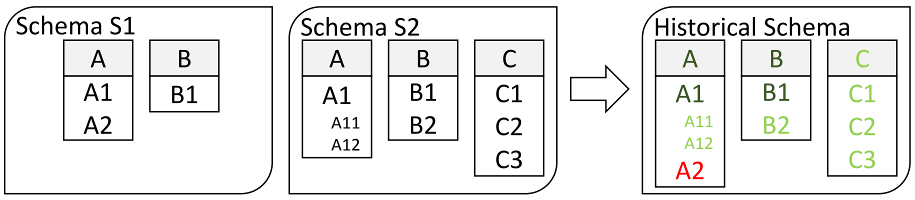


Figure 8.14: Example of schema evolution and the corresponding historical schema.

schema evolution and the corresponding colourized historical schema is given in Figure 8.14.

Exploitation

Exploiting the historical schema can facilitate the understanding of the schema evolution, and it can allow developers to spot potentially severe runtime errors or irretrievable data losses, together with the related code locations.

Colourization benefits. The color assigned to an object gives indications about its liveness. While the red color is assigned to objects which do not belong to the latest schema version, it does not necessarily represent a deleted object; it might only represent a field/collection that is no longer accessed in the latest source code version but that still exists in the database schema. Moreover, even if a red object is actually deleted from the schema, it does not ensure that there is no legacy data linked to this object. A red object only represents a soft warning to make developers aware of *potentially* outdated entities which should be either migrated or kept in mind for future developments.

Type mismatch detection. As previously explained, a data type change may cause error/crash if not properly managed by developers. Detecting the occurrence of a data type change is made possible by the historical schema and its meta-information. Indeed, since we can know the data type(s) of any field at any system version, we can easily detect a data type change.

Renaming detection. When a field or a collection is renamed or moved, developers need to keep it in mind for future developments. Indeed, similarly to a data type change, the legacy data have to be managed. Therefore, our automatic approach supports the identification of implicit (field/collection) renamings. The detection algorithm (detailed in Section 3.2.5) is based on different comparison criteria (e.g., name similarity, the field type similarity, etc.).

Data corruption/loss detection. Since a schema-less data store does not require an explicit schema, no verification before inserting a new record in the database is done, which might cause a data corruption/loss (e.g., accidentally removing/erasing stored data). Analyzing the historical schema of a system can help detecting such

Studied Period	#Versions	#Fields	#Accesses
29/11/2014 → 06/12/15	303	43 → 94	95 → 237

Table 8.4: Information concerning Tilos Radio.

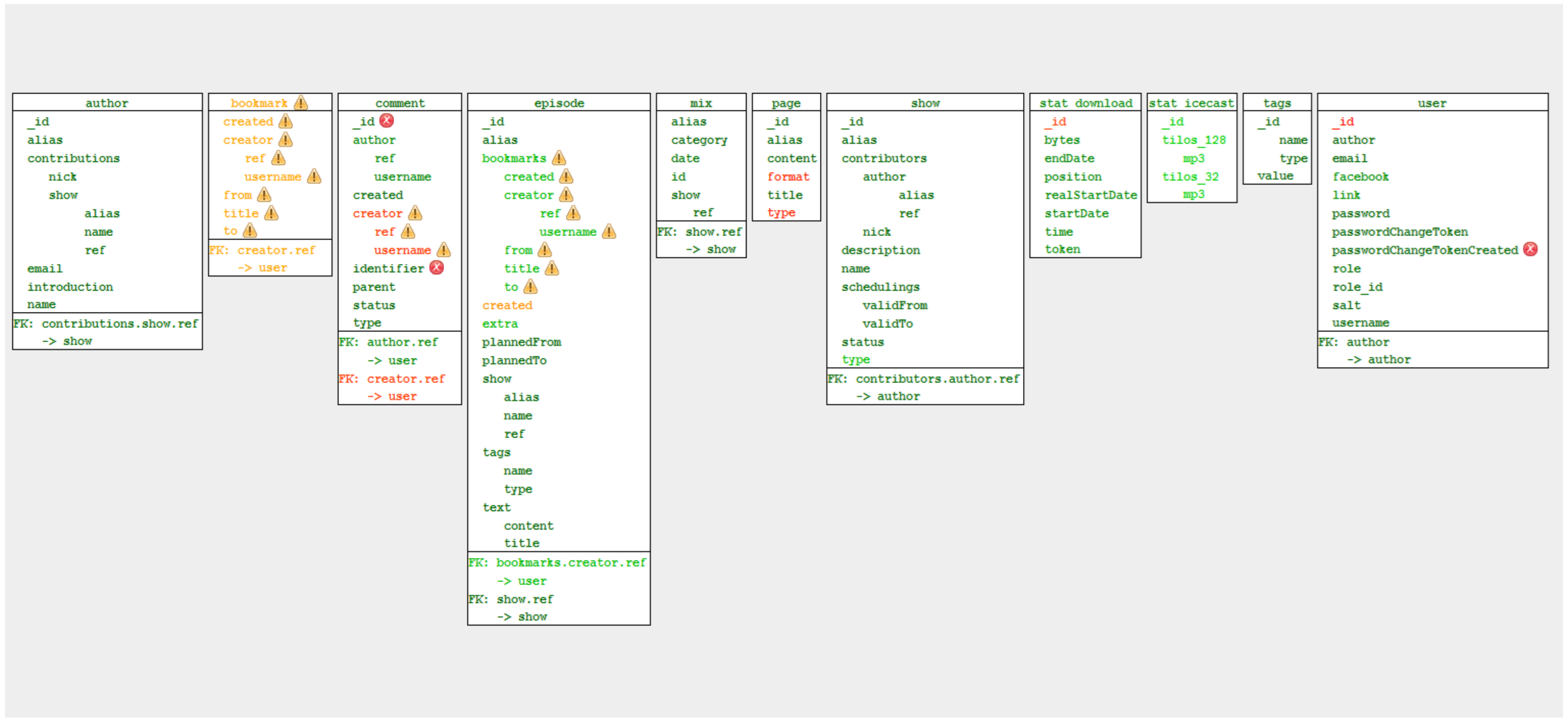
situations. For instance, a field receiving several different data types at a same version can denote a potential error that might lead to a data corruption/loss.

8.4.2 Early Evaluation

We applied our approach to a particular subject system containing the backend services of the *Tilos Radio*[6]. The Tilos Radio is a community, non-profit radio station in Budapest, Hungary. The versioning system of this project has a two-year history[7]. Since the introduction of MongoDB in the project, 303 versions of the system were committed on a period of one year. Its two-year evolution history and the use of MongoDB make Tilos Radio a good candidate for evaluating our historical analysis approach.

We applied our schema extraction approach to each version and computed the corresponding historical schema. Table 8.4 describes the evolution of Tilos. From the introduction of MongoDB until the latest version, the number of fields has more than doubled (from 43 to 94 fields). The number of accesses to the data store detected by our approach has also increased between the initial version and the latest one (from 95 to 237 accesses).

[6]https://tilos.hu/page/english
[7]https://github.com/tilosradio/web2-backend/

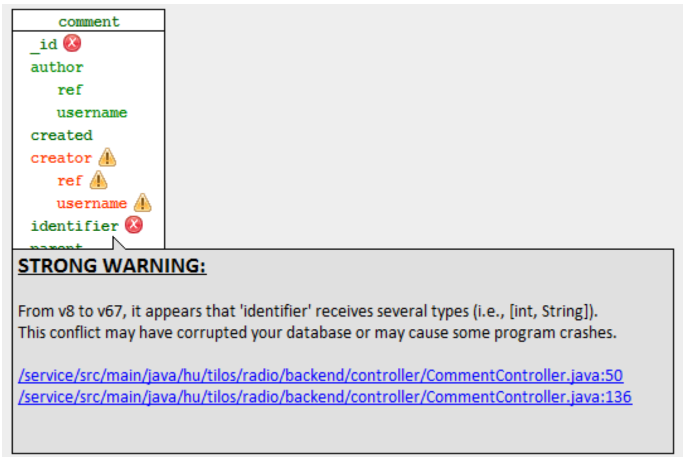


```
author
  _id
  alias
  contributions
    nick
    show
      alias
      name
      ref
  email
  introduction
  name
FK: contributions.show.ref
  -> show

bookmark ⚠
  created ⚠
  creator ⚠
    ref ⚠
    username ⚠
  from ⚠
  title ⚠
  to ⚠
FK: creator.ref
  -> user

comment
  _id ⊗
  author
    ref
    username
  created
  creator ⚠
    ref ⚠
    username ⚠
  identifier ⊗
  parent
  status
  type
FK: author.ref
  -> user
FK: creator.ref
  -> user

episode
  _id
  alias
  bookmarks ⚠
    created ⚠
    creator ⚠
      ref ⚠
      username ⚠
    from ⚠
    title ⚠
    to ⚠
  created
  extra
  plannedFrom
  plannedTo
  show
    alias
    name
    ref
  tags
    name
    type
  text
    content
    title
FK: bookmarks.creator.ref
  -> user
FK: show.ref
  -> show

mix
  alias
  category
  date
  id
  show
    ref
FK: show.ref
  -> show

page
  _id
  alias
  content
  format
  title
  type

show
  _id
  alias
  contributors
    author
      alias
      ref
    nick
  description
  name
  schedulings
    validFrom
    validTo
  status
  type
FK: contributors.author.ref
  -> author

stat download
  _id
  bytes
  endDate
  position
  realStartDate
  startDate
  time
  token

stat icecast
  _id
  tilos_128
    mp3
  tilos_32
    mp3

tags
  _id
    name
    type
  value

user
  _id
  author
  email
  facebook
  link
  password
  passwordChangeToken
  passwordChangeTokenCreated ⊗
  role
  role_id
  salt
  username
FK: author
  -> author
```

Figure 8.15: Historical schema of Radio Tilos displayed by our visualization tool.

The historical schema, shown in Figure 8.15, is visualized by our visualization tool. The latter provides developers with automatic reports about what happened in the system past. Icons warn developers of particular past events; clicking on those icons allows one to display automatic reports about those events. *Error* icons report on past events that might cause program crashes or data corruption. *Warning* icons aim to make developers aware of past events that should be considered for future developments (e.g., renamed fields/collections). Warning icons report on **soft** warnings while error icons report on **strong** warnings.

By analyzing this historical schema and the automatic reports processed by our tool, we made interesting observations concerning the schema evolution of this subject system.



STRONG WARNING:

From v8 to v67, it appears that 'identifier' receives several types (i.e., [int, String]).
This conflict may have corrupted your database or may cause some program crashes.

/service/src/main/java/hu/tilos/radio/backend/controller/CommentController.java:50
/service/src/main/java/hu/tilos/radio/backend/controller/CommentController.java:136

Figure 8.16: Automatic report emitting a strong warning concerning the type of the `comment.identifier` field.

The first automatic report (depicted in Figure 8.16), generated by clicking on the `comment.identifier` field, emits a strong warning. This strong warning informs users that a conflict with the type of the `comment.identifier` field happened from version 8 to version 67; indeed, it appeared that this field received several types (integer and string) during this period. This conflict might cause program crashes if the program attempts to load legacy integer entities and to store them in string parameters. The automatic report provides users with direct links to the source code showing the code locations where this conflict appeared[8]. Listings 8.12 and 8.13 show, respectively, the save of string and integer values in the same system version.

Another interesting observation is the automatic detection (see report depicted in Figure 8.17) of a particular renaming occurred at version 198: the `bookmarks` collection was moved (renamed) and became a *compound* field of the `episode` collection, as illustrated by Listings 8.14 and 8.15 showing the changes in the source code, respectively, before and after the renaming[9]. Thus, the remaining legacy

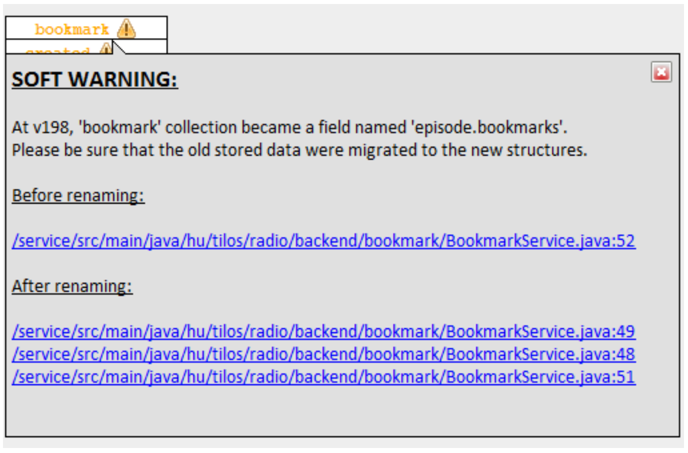
[8]Integer value:http://bit.ly/2i8BUFl. String value:http://bit.ly/2jycscv
[9]Before move:http://bit.ly/2d35A1b. After move:http://bit.ly/2dlzLQw

```java
DB db;
public List<CommentData> list(CommentType type, String id) {
  BasicDBObject query = new BasicDBObject();
  query.put("identifier", id);
  DBCursor comments = db.getCollection("comment").find(query);
  ...
}
```

Listing 8.12: Save of `identifier` as a string in version 67.

```java
DB db;
public CreateResponse create(CommentType type, int id, CommentToSave data) {
  BasicDBObject comment = modelMapper.map(data, BasicDBObject.class);
  comment.put("identifier", id);
  ...
  db.getCollection("comment").insert(comment);
  ...
}
```

Listing 8.13: Save of `identifier` as an integer in version 67.



SOFT WARNING:

At v198, 'bookmark' collection became a field named 'episode.bookmarks'.
Please be sure that the old stored data were migrated to the new structures.

Before renaming:

/service/src/main/java/hu/tilos/radio/backend/bookmark/BookmarkService.java:52

After renaming:

/service/src/main/java/hu/tilos/radio/backend/bookmark/BookmarkService.java:49
/service/src/main/java/hu/tilos/radio/backend/bookmark/BookmarkService.java:48
/service/src/main/java/hu/tilos/radio/backend/bookmark/BookmarkService.java:51

Figure 8.17: Before and after moving `bookmark` in `episode` at version 198.

entities of the outdated `bookmark` collection should be managed accordingly by developers.

Figure 8.18 reports on a potential data loss occurred in the system past. Our approach automatically spotted a potential data loss due to a misuse of the `user.passwordChangeTokenCreated` field. Indeed, developers assigned a wrong value to this field, which overwrites the correct value, as depicted in Listings 8.16 and 8.17 showing the potential loss before and after fixing the mistake[10]. This mistake stayed unfixed during 20 system versions (from version 0 to version 19), what could represent an important data loss since the correct values to store in the

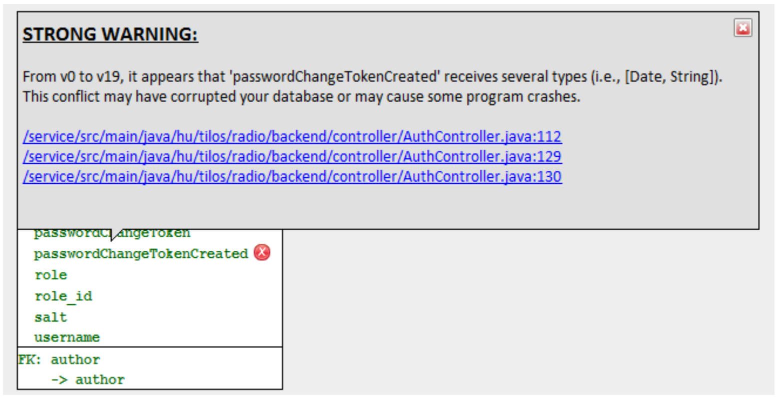
[10]Before fix:http://bit.ly/2cxI1A3. After fix:http://bit.ly/2cPbQII

```
Db db;
public CreateResponse create(Session session, String episodeId, BookmarkToSave
      bookmarkToSave) {
  BasicDBObject episodeSelector = new BasicDBObject("_id", new ObjectId(episodeId));
  BasicDBObject bookmark = new BasicDBObject();
  bookmark.put("from", bookmarkToSave.getFrom());
  ...
  db.getCollection("bookmark").insert(bookmark);
  ...
}
```

Listing 8.14: Insertion of a new bookmark into the `bookmark` collection before its renaming.

```
DB db;
public CreateResponse create(Session session, String episodeId, BookmarkToSave
      bookmarkToSave) {
  BasicDBObject episodeSelector = new BasicDBObject("_id", new ObjectId(episodeId));
  DBObject episode = db.getCollection("episode").findOne(episodeSelector);
  BasicDBObject bookmark = new BasicDBObject();
  bookmark.put("from", bookmarkToSave.getFrom());
  ...
        if (episode.get("bookmarks") == null) {
            episode.put("bookmarks", new BasicDBList());
        }
  ((BasicDBList) episode.get("bookmarks")).add(bookmark);
  db.getCollection("episode").update(episodeSelector, episode, true, false);
  ...
}
```

Listing 8.15: Insertion of a new bookmark into the `episode.bookmark` field after its renaming at version 176.


STRONG WARNING:

From v0 to v19, it appears that 'passwordChangeTokenCreated' receives several types (i.e., [Date, String]).
This conflict may have corrupted your database or may cause some program crashes.

/service/src/main/java/hu/tilos/radio/backend/controller/AuthController.java:112
/service/src/main/java/hu/tilos/radio/backend/controller/AuthController.java:129
/service/src/main/java/hu/tilos/radio/backend/controller/AuthController.java:130

Figure 8.18: Loss of the value to store in `passwordChangeTokenCreated`.

database are definitely lost. Further analysis revealed that the wrong value should have been assigned to another field of the `user` collection.

```java
DB db;
private Response generateToken(PasswordReset passwordReset) {
  DBObject user = db.getCollection("user").findOne(new BasicDBObject("email",
       passwordReset.getEmail()));
  ...
  String token = authUtil.generateSalt();
  user.put("passwordChangeTokenCreated", new Date());
  user.putpasswordChangeTokenCreatedtoken);
  db.getCollection("user").update(new BasicDBObject("username", user.get("username")),
       user);
  ...
}
```

Listing 8.16: Loss of the value to store in `passwordChangeTokenCreated`.

```java
DB db;
private Response generateToken(PasswordReset passwordReset) {
  DBObject user = db.getCollection("user").findOne(new BasicDBObject("email",
       passwordReset.getEmail()));
  ...
  String token = authUtil.generateSalt();
  user.put("passwordChangeTokenCreated", new Date());
  user.put("passwordChangeToken", token);
  db.getCollection("user").update(new BasicDBObject("username", user.get("username")),
       user);
  ...
}
```

Listing 8.17: Fix of the data loss related to `passwordChangeTokenCreated`.

8.4.3 Discussion

In summary, this Section presents two main novel contributions: (1) a static analysis approach, specifically designed for Java systems using MongoDB, which extracts the NoSQL database schema from the application source code (as unique information source) by exploring the call graph and the intraprocedural control-flow of the application; (2) a historical analysis which helps developers to understand the schema evolution and allows the automatic detection of potential errors and data losses.

We applied this approach to the whole history of a subject system and we computed the so-called historical schema. We finally showed how analyzing the past of a system, by using this historical schema, can be useful to understand the present version and to ease future developments. In particular, our approach automatically detects and warns developers about potential risks, such as past data structure changes, data type mismatches and data losses.

Limitations

The schema extraction phase of our approach may be affected by several limitations. Our tool approach is specifically designed for systems using the MongoDB Java driver to communicate with the database. While Java is the most popular programming language today [TIOBE Programming Community Index, 2017] and MongoDB is the most used NoSQL database systems [DB-Engines Ranking, 2017], the presented approach could be extended to other languages and database systems.

Secondly, since the schema extraction partially relies on our static analysis approach (presented in Section 4.2) exploiting the intraprocedural control-flow of the application, its static nature is mainly at the basis of some limitations (as further discussed in Section 4.3.3).

In the future, we intend to conduct empirical studies on a large set of systems to analyze how developers evolve NoSQL databases in practice and to further study the entropy introduced by this evolution.

8.5 Concluding Remarks

In this Chapter, we reused several principles and approaches presented in the previous chapters. We showed their usefulness by applying them to three other fields:

(a) We adapted our static analysis solution presented in Chapter 4 for a typical concept location problem for data-intensive systems: "where was this query executed?". We adapted our static technique for identifying the exact source code location(s) from where a given SQL query was sent to the database server.

(b) We devised a process for reengineering legacy information systems with respect to establishing referential integrity constraints, incorporating, combining and extending existing reengineering methods. In particular, analyzing the database usage within the source code can help at finding implicit referential integrity constraints; hence, we use our static analysis approach (presented in Chapter 4) to recover database accesses from the source code.

(c) We presented an automatic approach that aims at understanding schema evolution in NoSQL data stores. We presented two main novelties:

   - an automatic approach that infers the database schema of a schema-less NoSQL data store by analyzing the database usage within the application source code. We adapted our static analysis technique (presented in Chapter 4) to MongoDB applications.
   - the application of this approach to the whole system history in order to generate the so-called historical schema (defined in Chapter 3); the latter allows understanding schema evolution and preventing errors and data losses.

Conclusion and Future Directions

Summary of the Contributions

At the beginning of this thesis, we identified a set of research questions. The below Table summarizes the targeted research questions of each chapter.

	Research Questions		
Chapters	RQ1	RQ2	RQ3
Chapter 3	✓		
Chapter 4		✓	
Chapter 5	✓		
Chapter 6			✓
Chapter 7			✓
Chapter 8	✓		

Distribution of the research questions targeted by each thesis chapter.

RQ1: How can history analysis of a DISS support the actual maintenance of the system?

In Chapters 3, 5 and 8, we presented analysis techniques which can be applied to support the actual maintenance of a DISS.

In Chapter 3, we presented a historical analysis approach that allows us to analyze the evolution history of a given database schema. The method is based on the automated derivation of a historical schema, that includes all the schema objects involved in the entire lifetime of the database, each annotated with historical and temporal information. By conducting a study on a complex real-life system concerned by a migration project, we **showed** the benefits of such a history analysis to **understand the current database structures**, which is a prerequisite to the migration and the future developments. Among others, we detected the presence of superseded structures and understood the reasons of their presence in the current system version. This historical analysis also allowed us to understand the role of some recently created large-scale tables. In addition, we also analyzed developers' activities on the database schema and identified the database specialists.

In Chapter 5, we proposed a second historical analysis approach allowing us to understand, at a fine-grained level, how systems evolve over time, how the database and code co-evolve and how several technologies may co-exist into the same system. In particular, our approach focused on the evolution of several system artefacts, namely the source code, the database schema, the database usage and the ORM usage. By conducting a study on three real-life systems, we **showed** the benefits of our historical analysis approach to understand how the program and database have co-evolved over time and **how this evolution has led to the current system state**. We studied the co-existence of several access technologies within a same system and understood the objectives of introducing a new technology in a project (e.g., complementing or replacing the existing one). It also allowed us to detect current bad practices when using some technologies and to understand the historical reasons. All those observations can guide future developments and facilitate decision making.
Although this approach requires some technology-dependent tools to extract the database and ORM usage, the conceptual model on which our historical analysis relies could be totally technology-independent with minor adaptations.

In Chapter 8, we presented a third historical analysis approach that aims at understanding schema evolution in NoSQL data stores. This approach presented two main contributions, namely, (1) an automatic approach that infers the database schema of a schema-less NoSQL data store by analyzing the application source code and (2) the application of this approach to the whole system history in order to derive the historical schema. By conducting a study on a real-life system, we **showed** the benefits of our approach. In particular, this historical analysis allowed us to **detect potentially critical code locations** that could lead to runtime errors or data losses, due to historical reasons.

RQ2: How to automatically analyze and extract the communication between application programs and the database in a dynamic DISS?

In Chapter 4, we **demonstrated** that it is possible to automatically detect and recover database accesses within the source code by use of a **static program analysis technique**. This analysis approach permits the automatic recovery of SQL queries which are **dynamically** constructed in the code or which are partially/fully hidden because generated by the ORM layer. By conducting a study on three real-life systems, we also **showed** that this static approach could reach **good results**, with 71.5 - 99% of successfully extracted queries and 87.9 - 100% of valid queries.

This technique is specifically designed for Java systems (the most popular programming language today [TIOBE Programming Community Index, 2017]) and targets three of the most popular Java access technologies, i.e., JDBC, Hibernate and JPA. The automated detection and recovery of the database usage in the source code is a first essential step towards the implementation of an approach supporting the co-evolution between the programs and the database.

RQ3: To what extent can we support program-database co-evolution in dynamic and heterogeneous DISS?

In Chapter 6, we **showed** that our proposed **what-if analysis approach** constitutes a good support of program-database co-evolution in dynamic and heterogeneous DISS. This approach 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 **automatic recommendations** to developers about where and how they should propagate, **at the line of code level**, the schema changes to the source code. Based on the study of three real-life systems, we showed that our what-if analysis approach could reach **very good results**, with 99% of correct recommendations when applied to a randomly selected schema changes.

In Chapter 7, we **showed** that it is possible to support program-database co-evolution in dynamic and heterogeneous DISS by use of our visual analyzer, DAHLIA 2.0. DAHLIA 2.0 implements a city metaphor that allows an intuitive analysis of the database usage of dynamic and heterogeneous systems by **visualizing the links** between the source code and the database. DAHLIA 2.0 allows developers to assess the future impact of any change on the system by *highlighting* the system part that will be impacted by that change. We then showed the **applicability** of our tool to eleven real-life systems. We finally proved its **suitability** by conducting a controlled experiment; we observed that DAHLIA 2.0 can influence completion time and correctness of performing comprehension tasks pertaining to the program-database communication, with a decrease of time to achieve the tasks (-41.99%) and a higher level of correctness (+7.33%).
Our tool can deal with systems using several database access technologies together like JDBC, Hibernate and JPA. Nevertheless, DAHLIA 2.0 and its visualization principles could be easily used to visualize DISS written in other programming languages than Java and using other access technologies than JDBC, Hibernate and JPA. Indeed, unlike the data extraction process, the visualization is technology-independent.

Future Directions

To conclude this thesis, we highlight interesting perspectives for future research.

Extending the scope of considered programming languages and technologies. Although most of the presented methodologies are generic, some tool-supported approaches that we implemented in this thesis are specifically designed to support the evolution of Java systems using access technologies such as JDBC, Hibernate and JPA. This technological choice was guided by some surveys (e.g., the TIOBE index [TIOBE Programming Community Index, 2017]) revealing that Java is the most popular (used) programming language in the world. In particular, Goeminne *et al.* [Goeminne and Mens, 2015] carried out a large-scale empirical study and revealed that those three particular Java database access technologies (JDBC, Hibernate and JPA) were among the most popular. However, other popular programming languages (e.g., C, C++, C#, PHP, ...) and access technologies/frameworks (e.g., Spring, iBATIS, ...) could be considered.

Taking into account data in the DAHLIA visualization.
Chapters 3 and 7 presented our visualization tool, DAHLIA (and its extension DAHLIA 2.0). This tool makes use of the city metaphor to visualize (historical) information about the database schema. This city metaphor (borrowed from CodeCity [Wettel and Lanza, 2008a]) exploits several dimensions to visualize this information, i.e., the building width/height/color/packaging. Another visual metric could be to consider data stored in each database table. Indeed, the number of records in a table could be visualized and affected to the building height. The taller the table, the higher the number of records stored in the table. It would allow users to assess the required effort to migrate the stored data in case of a database refactoring (e.g., table split/merge/etc.).

Automating the schema change propagation process. In the context of the co-evolution of databases and programs, *impact analysis* is an important process. Indeed, an in-depth analysis of the impact of a future database schema change is required to precisely assess the program adaptation effort; in Chapters 6 and 7, we respectively presented (1) an automatic approach making recommendations to programmers about where and how to propagate a simulated schema change to the source code, and (2) DAHLIA 2.0 allowing developers to visualize the interactions between the application programs and the database. We consider that those tool-supported approaches constitute a preliminary phase to reach our ultimate objective, namely contributing to (partially) automate the database schema change propagation process itself, via source code transformation techniques.

Automating conceptual re-documentation of database usage in source code.
Linares-Vásquez et al. [Linares-Vasquez et al., 2015] presented an empirical study (implying 3.1K open-source Java systems) that reveals that a large proportion of database-accessing methods is completely undocumented. Later [Linares-Vásquez et al., 2016], the authors presented *DBScribe*. DBScribe statically analyzes the code and database schema to detect database usages; it then automatically generates natural language documentation at source code method level that describes database usage for a given system. However, the proposed static analysis approach has critical limitations while (1) extracting the database usage - i.e., it only intraprocedurally reconstructs the access and thus misses essential inter-procedural information - and (2) generating automatic documentation - i.e., the used templates to generate documentation are at the *logical* level and mainly in the simple form *"this instruction inserts the <attr> attributes into table <table>"*. An extension of our work could be to devise an automatic approach that would consist of defining the *conceptual*

interpretation of SQL queries (recovered by our static analysis approach) in terms of a domain-specific, platform-independent model. Based on the database conceptual schema, this approach would generate a *high-level* documentation (e.g., *this method removes all the orders placed by a given customer from the database*). This possible future direction is inspired by the work of Noughi et al. [Noughi and Cleve, 2015].

Conducting an empirical study to detect the most frequent SQL query anti-patterns. Several authors have listed the common anti-patterns in SQL queries [Karwin, 2010; Phil Factor, 2010]. Those SQL code smells can have impact on the database performance or alter the program behaviour. Hence, detecting the presence of such SQL code smells is sometimes indispensable to improve the database performance or avoid unexpected program behaviours at runtime. An interesting future direction would be to extend our static analysis approach (presented in Chapter 4) to (1) extract the SQL queries from the source code and (2) automatically spot the presence of anti-patterns. In this way, we could conduct an empirical study on hundreds/thousands of open-source systems in order to automatically detect the most frequent encountered SQL query anti-patterns and study their longevity/impact in a project.

Supporting the migration of relational systems towards NoSQL. NoSQL technologies can face some limitations of traditional relational database technology (e.g., improving the response time, handling frequent data writes, etc.). The appearance of NoSQL technologies raises new challenges in software evolution. For instance, replacing legacy relational databases by NoSQL databases is a challenge very similar to this thesis topic. Indeed, detecting/recovering the SQL communication between programs and database, and then adapting the legacy SQL accesses to the new NoSQL structures is challenging, especially in presence of dynamic and heterogeneous systems. A real-world example is a current project conducted by the OSCAR developers, the main objective of which is to (partially) migrate the relational database to a NoSQL model to improve performance. Therefore, automated support to assist developers in this task of migrating relational systems towards NoSQL systems would be a precious asset.

Studying the co-existence of relational and NoSQL databases within a same system, and supporting their co-evolution. The emergence of NoSQL technologies raises new challenges while migrating existing (relational) systems towards NoSQL platforms. However, NoSQL technologies are not necessarily intended to fully replace relational databases. Instead, NoSQL technologies can be seen as a complementary paradigm. The presence of those two different paradigms, i.e., NoSQL and relational, in a same system can pose new problems to ensure their co-existence and co-evolution in a consistent manner. In particular, a novel future research direction could be to study in practice how those technologies co-exist and co-evolve together in a same system. Moreover, automated support to facilitate this co-evolution would be helpful.

Designing automated model-driven engineering for NoSQL systems. Although NoSQL databases are so-called schema-less, it does not mean that data are not structured. Several authors have discussed the need for data-model approaches while designing NoSQL databases [Olivera et al., 2015; de Lima and dos Santos Mello, 2015; Atzeni, 2016]. We claim that an automated model-driven engineering is primordial to design NoSQL systems in a disciplined manner. Inspired by the database engineering process, our proposed methodology would produce different models, from a technology-independent (e.g., the conceptual schema) to a technology-specific (e.g., the physical schema) point of view. Similar to the relational database design process, the conceptual schema would be a conceptual representation (in a technology-independent manner) of the data, in terms of entities, relationships and attributes. The physical schema would describe the technical constructs of the database (e.g., a document/key-value/graph representation). This representation would directly depend on the chosen NoSQL technology (more than 50 NoSQL technologies exist [Stonebraker, 2011]) and would also allow the definition of some technical optimizations (e.g., denormalization, data redundancy, ...). In addition, mappings between objects of both schemas would be established; for instance, a concept/attribute/relationship of the conceptual schema can be mapped to one or several collections/fields of the physical schema that models a document store (e.g., MongoDB). In this way, one keeps traceability on implicit integrity constraints imposed on the stored data (e.g., data redundancy, foreign keys, etc.). Moreover, a potential useful asset of our proposed methodology would consist in keeping track of different versions of data objects by use of version-based models. The final step of this NoSQL database design process could be the automatic generation of DAO classes (in the target programming language and NoSQL framework), that could be automatically regenerated in case of changes in the physical schema. The benefits of such an engineering process would be to propose a disciplined manner to design NoSQL databases, and to have an up-to-date documentation facilitating future developments.

APPENDIX A

Controlled Experiment for the Empirical Evaluation of DAHLIA 2.0

In Chapter 7, we presented DAHLIA 2.0 and then conducted a controlled experiment to evaluate the suitability of the approach. As mentioned in Section 7.4, we decided to provide the control group of our experiment with data tables containing the metrics required to solve the tasks. Those data were stored in 3 Excel sheets. The structure of each sheet is described in Figures A.1, A.2 and A.3.

The first sheet depicts a two-dimensional array, where (1) the horizontal axis contains the set of tables defined in the database schema of the subject system, and (2) the vertical axis contains the set of Java files of the subject system. This sheet represents a binary matrix where the couple $(File_i, Table_j) = 1$ if $File_i$ accesses table $Table_j$ (0, otherwise). This matrix allows students to know which tables are accessed by which files.

The second sheet also depicts a two-dimensional array, where (1) the horizontal axis still contains the set of tables defined in the database schema of the subject system, and (2) the vertical axis contains the lines of code, grouped by Java file, that access the database. In addition, the $Techno$ column specifies the used database access technology. This sheet contains a binary matrix where the couple $(File_{i:j}, Table_k) = 1$ if line j of $File_i$ accesses table $Table_k$ (0, otherwise). This matrix allows students to know which lines of code of which files access which tables (and by use of which technology).

The third sheet also depicts a two-dimensional array, where (1) the horizontal axis still contains the set of tables defined in the database schema of the subject system, and (2) the vertical axis contains the three access technologies, i.e., JDBC, Hibernate and JPA. This sheet contains a binary matrix where the couple $(techno, Table_j) = 1$ if the access technology $techno$ accesses table $Table_j$ (0, oth-

	Table_1	Table_2	Table_3	...	Table_m
File_1	0	1	1	...	0
File_2	1	0	0	...	0
File_3	0	0	0	...	1
...					
File_m	1	1	0	...	1

Figure A.1: Structure of the first Excel sheet.

	Techno	Table_1	Table_2	Table_3	...	Table_m
$\text{File}_{1:12}$	JDBC	0	1	0	...	0
$\text{File}_{1:51}$	HIB	0	0	1	...	0
$\text{File}_{2:21}$	JDBC	1	0	0	...	0
$\text{File}_{3:8}$	HIB	0	0	0	...	1
$\text{File}_{3:81}$	HIB	0	0	0	...	1
$\text{File}_{3:125}$	JPA	0	0	0	...	1
...						
$\text{File}_{m:x}$	JDBC	1	1	0	...	1

Figure A.2: Structure of the second Excel sheet.

Techno	Table_1	Table_2	Table_3	...	Table_m
JDBC	1	1	0	...	1
HIB	0	0	1	...	1
JPA	0	0	0	...	1

Figure A.3: Structure of the third Excel sheet.

erwise). This matrix allows students to know which technologies access which database tables.

The raw measurements of completion time and correctness performed during the experiment are shown in Table A.1.

Student	**Tool**	**Education**	**Time (seconds)**	**Score**
1	DAHLIA	M	568	7
2	DAHLIA	M	600	8
3	DAHLIA	M	384	8
4	DAHLIA	M	444	7
5	DAHLIA	M	329	8
6	DAHLIA	M	809	8
7	DAHLIA	E	488	8
8	DAHLIA	E	407	6
9	DAHLIA	E	327	8
10	DAHLIA	E	600	8
11	DAHLIA	E	317	8
12	DAHLIA	E	635	8
13	DAHLIA	E	424	7
14	DAHLIA	E	678	6
15	DAHLIA	E	455	8
16	DAHLIA	E	294	7
17	DAHLIA	E	408	8
18	DAHLIA	E	243	8
19	DAHLIA	E	772	6
20	DAHLIA	B	488	8
21	DAHLIA	B	310	7
22	DAHLIA	B	405	8
23	DAHLIA	B	360	8
24	DAHLIA	B	357	8
25	DAHLIA	B	465	8
26	EXCEL	M	659	8
27	EXCEL	M	831	6
28	EXCEL	M	643	8
29	EXCEL	M	664	7
30	EXCEL	M	585	7
31	EXCEL	E	971	6
32	EXCEL	E	703	8
33	EXCEL	E	591	8
34	EXCEL	E	868	7
35	EXCEL	E	570	5
36	EXCEL	E	461	7
37	EXCEL	E	895	7
38	EXCEL	E	793	6
39	EXCEL	E	697	8
40	EXCEL	E	632	8
41	EXCEL	B	1062	5
42	EXCEL	B	1361	7
43	EXCEL	B	577	7
44	EXCEL	B	1058	8
45	EXCEL	B	1085	7
46	EXCEL	B	698	8
47	EXCEL	B	953	6
48	EXCEL	B	987	8

Table A.1: Measured completion times (in seconds) and scores (1-8).

Bibliography

Alalfi, M. H., Cordy, J. R., and Dean, T. R. (2009). WAFA: fine-grained dynamic analysis of web applications. In **Proceedings of the 11th IEEE International Symposium on Web Systems Evolution, WSE 2009, 25-26 September 2009, Edmonton, Alberta, Canada**, pages 141–150.

Allamanis, M. and Sutton, C. (2013). Mining source code repositories at massive scale using language modeling. In **Proceedings of the 10th IEEE Working Conference on Mining Software Repositories, MSR 2013, 18–26 May 2013, San Francisco, CA, USA**, pages 207–216.

Allen, F. E. (1970). Control flow analysis. In **Proceedings of a Symposium on Compiler Optimization, 27–28 July 1970, Illinois, USA**, pages 1–19. ACM.

Antwerp, M. V. and Madey, G. R. (2010). The importance of social network structure in the open source software developer community. In **Proceedings of the 43rd IEEE Hawaii International Conference on System Sciences, HICSS 2010, 5–8 January 2010, Honolulu, HI, USA**, pages 1–10.

Atzeni, P. (2016). Data modelling in the nosql world: A contradiction? In **Proceedings of the 17th International Conference on Computer Systems and Technologies, CompSysTech 2016, 22–24 June 2016, Palermo, Italy**, pages 1–4. ACM.

Brink, H. v. d., Leek, R. v. d., and Visser, J. (2007). Quality assessment for embedded SQL. In **Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2007, September 30 – October 1 2007, Paris, France**, pages 163–170.

Canfora, G., Cerulo, L., Cimitile, M., and Di Penta, M. (2011). Social interactions around cross-system bug fixings: The case of freebsd and openbsd. In **Proceedings of the 8th Working Conference on Mining Software Repositories, MSR 2011, 21–22 May 2011 ,Waikiki, Honolulu, HI, USA**, pages 143–152. ACM.

Chang, S.-K., Deufemia, V., Polese, G., and Vacca, M. (2007). A logic framework to support database refactoring. In **Proceedings of the 18th International Conference on Database and Expert Systems Applications, DEXA 2007, 3–7 September 2007, Regensburg, Germany**, pages 509–518. Springer.

Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., and Flora, P. (2016). Detecting problems in the database access code of large scale systems: An industrial experience report. In **Proceedings of the 38th International Conference on Software Engineering Companion, ICSE 2016, 14–22 May 2016, Austin, Texas, USA**, pages 71–80. ACM.

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. In **Proceedings of the 36th IEEE International Conference on Software Engineering, ICSE 2014, May 31 – June 7 2014, Hyderabad, India**, pages 1001–1012. ACM.

Chiang, R. H. L., Barron, T. M., and Storey, V. C. (1994). Reverse engineering of relational databases: extraction of an eer model from a relational database. **Data & Knowledge Engineering**, 12(2):107–142.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, 20(6):476–493.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2003). Precise analysis of string expressions. In **Proceedings of the 10th International Conference on Static Analysis, SAS 2003, 11–13 June 2003, San Diego, CA, USA**, pages 1–18. Springer-Verlag.

Clark, S. R., Cobb, J., Kapfhammer, G. M., Jones, J. A., and Harrold, M. J. (2011). Localizing SQL faults in database applications. In **Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, 6–10 November 2011, Washington, DC, USA**, pages 213–222.

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

Cleve, A. and Hainaut, J.-L. (2008). Dynamic analysis of sql statements for data-intensive applications reverse engineering. In **Proceedings of the 15th IEEE Working Conference on Reverse Engineering, WCRE 2008, 15–18 October 2008, Antwerp, Belgium**, pages 192–196.

Cleve, A., Henrard, J., and Hainaut, J.-L. (2006). Data reverse engineering using system dependency graphs. In **Proceedings of the 13th IEEE Working Conference on Reverse Engineering, WCRE 2006, 23–27 October 2006, Benevento, Italy**, pages 157–166.

Cleve, A., Mens, T., and Hainaut, J.-L. (2010). Data-intensive system evolution. **IEEE Computer**, 43(8):110–112.

Cleve, A., Noughi, N., and Hainaut, J.-L. (2012). Dynamic program analysis for database reverse engineering. In **Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, 3–9 July 2011, Braga, Portugal**, pages 297–321. Springer. LNCS.

Cohen, W. W., Ravikumar, P., and Fienberg, S. E. (2003). A comparison of string metrics for matching names and records. In **KDD Workshop on Data Cleaning and Object Consolidation**, pages 73–78.

Curino, C. A., Moon, H. J., Deutsch, A., and Zaniolo, C. (2013). Automating the database schema evolution process. **The VLDB Journal**, 22(1):73–98.

Curino, C. A., Moon, H. J., Ham, M. W., and Zaniolo, C. (2009). The prism workwench: Database schema evolution without tears. In **Proceedings of the 18th IEEE International Conference on Database and Expert Systems Applications, ICDE 2009, March 29 – April 2 2009, Shanghai, China**, pages 1523–1526.

Curino, C. A., Moon, H. J., Tanca, L., and Zaniolo, C. (2008). Schema evolution in Wikipedia - toward a web information system benchmark. In **Proceedings of the 10th International Conference on Enterprise Information Systems, ICEIS 2008, 12–16 June 2008, Barcelona, Spain**, pages 323–332.

D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. **Empirical Software Engineering.**, 17(4-5):531–577.

Darcy, D. P., Daniel, S. L., and Stewart, K. J. (2010). Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes. In **Proceedings of the 43rd Hawaii International International Conference on Systems Science, HICSS 2010, 5-8 January 2010, Koloa, Kauai, HI, USA**, pages 1–11.

DB-Engines Ranking (2017). http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2017-02-10.

de Lima, C. and dos Santos Mello, R. (2015). A workload-driven logical design approach for nosql document databases. In **Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services, iiWAS 2015, 11–13 December 2015, Brussels, Belgium**, pages 73:1–73:10. ACM.

Devanbu, P., Nagappan, N., Murphy, B., Bird, C., and Gall, H. (2009). Putting it all together: Using socio-technical networks to predict failures. In **Proceedings in the 20th IEEE International Symposium on Software Reliability Engineering, ISSRE 2009, 16–19 November 2009, Mysuru, India**, pages 109–119.

Di Lucca, G. A., Fasolino, A. R., and de Carlini, U. (2000). Recovering class diagrams from data-intensive legacy systems. In **Proceedings of the 16th IEEE International Conference on Software Maintenance, ICSM 2000, 11–14 October 2000, San Jose, CA, USA**, page 52.

Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In **Proceedings of the 7th International Colloquium on Automata, Languages and Programming, ICALP 1980, 14–18 July 1980, Noordweijkerhout, The Netherlands**, pages 169–181. Springer-Verlag.

Fernández-Ramil, J., Lozano, A., Wermelinger, M., and Capiluppi, A. (2008). Empirical studies of open source evolution. In **Software Evolution**, pages 263–288.

Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., and Tao, L. (2007). A static analysis framework for detecting sql injection vulnerabilities. In **Proceedings of the 31st IEEE International Computer Software and Applications Conference, COMPSAC 2007, 23–27 July 2007, Beijing, China**, pages 87–96.

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

Glass, R. L. (2001). Frequently forgotten fundamental facts about software engineering. **IEEE Software**, 18(3):112–111.

Göde, N. and Koschke, R. (2011). Frequency and risks of changes to clones. In **Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, 21–28 May 2011, Waikiki, Honolulu, Hawaii**, pages 311–320. ACM.

Godfrey, M. W. and Tu, Q. (2000). Evolution in open source software: A case study. In **Proceedings of the IEEE International Conference on Software Maintenance, ICSM 2000, 11–14 October 2000, San Jose, CA, USA**, pages 131–.

Goeminne, M., Decan, A., and Mens, T. (2014). Co-evolving code-related and database-related changes in a data-intensive software system. In **Proceedings of IEEE CSMR-WCRE 2014 Software Evolution Week, 3–6 February, Antwerp, Belgium**, pages 353–357.

Goeminne, M. and Mens, T. (2015). Towards a survival analysis of database framework usage in Java projects. In **Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, September 29 – October 1, 2015, Bremen, Germany**, pages 551–555.

Golfarelli, M., Rizzi, S., and Proli, A. (2006). Designing what-if analysis: Towards a methodology. In **Proceedings of the 9th ACM International Workshop on Data Warehousing and OLAP, DOLAP 2006, 10 November 2006, Arlington, Virginia, USA**, pages 51–58.

Gould, C., Su, Z., and Devanbu, P. (2004). Static checking of dynamically generated queries in database applications. In **Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, 23–28 May 2004, Edinburgh, Scotland, UK**, pages 645–654.

Grolinger, K. and Capretz, M. A. M. (2011). A unit test approach for database schema evolution. **Information and Software Technology**, 53(2):159–170.

Grosso, C. D., Penta, M. D., and de Guzmán, I. G. R. (2007). An approach for mining services in database oriented applications. In **Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Software Evolution**

in Complex Software Intensive Systems, CSMR 2007, 21-23 March 2007, Amsterdam, The Netherlands, pages 287–296.

Grubbs, F. E. (1950). Sample criteria for testing outlying observations. **The Annals of Mathematical Statistics**, 21(1):27–58.

Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In **Proceedings of the 32th International Conference on Software Engineering, ICSE 2010, 2–8 May 2010, Cape Town, South Africa**, pages 495–504. ACM.

Hainaut, J.-L. (2002). **Introduction to Database Reverse Engineering**. Institut d'Informatique - LIBD.

IEEE (1999). **IEEE Standard for Software Maintenance, IEEE Std 1219-1998**, volume 2. IEEE Press.

International Standards Organisation (ISO) (1999). **Standard 14764 on Software Engineering - Software Maintenance**. ISO/IEC.

Javid, M. A. and Embury, S. M. (2012). Diagnosing faults in embedded queries in database applications. In **Proceedings of the 15th International Conference on Extending Database Technology and Database Theory, EDBT/ICDT 2015, 26–30 March 2012, Berlin, Germany**, pages 239–244. ACM.

JBOSS Hibernate (2017). http://hibernate.org/orm/what-is-an-orm/. Accessed: 2017-05-09.

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

Karwin, B. (2010). **SQL Antipatterns: Avoiding the Pitfalls of Database Programming**. Pragmatic Bookshelf, 1st edition.

Khomh, F., Di Penta, M., Gueheneuc, Y.-G., and Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. **Empirical Software Engineering**, 17(3):243–275.

Kildall, G. A. (1973). A unified approach to global program optimization. In **Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1973, 1–3 October 1973, Boston, Massachusetts, USA**, pages 194–206.

King, J. C. (1976). Symbolic execution and program testing. **Communications of the ACM**, 19(7):385–394.

Lanza, M., Marinescu, R., and Ducasse, S. (2005). **Object-Oriented Metrics in Practice**. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. **Proceedings of IEEE**, 68(9):1060–1076.

Lehman, M. M. (1984). On understanding laws, evolution, and conservation in the large-program life cycle. **Systems and Software**, 1:213–221.

Lehman, M. M. (1996). Laws of software evolution revisited. In **Proceedings of the 5th European Workshop on Software Process Technology, EWSPT 1996, 09-11 October 1996, Nancy, France**, pages 108–124.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. In **Proceedings of the 4th IEEE International Software Metrics Symposium, METRICS 1997, 5–7 November 1997, Albuquerque, NM, USA**, pages 20–.

Lin, D.-Y. and Neamtiu, I. (2009). Collateral evolution of applications and databases. In **Proceedings of Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, 24–25 August 2009, Amsterdam, The Netherlands**, pages 31–40. ACM.

Linares-Vasquez, M., Li, B., Vendome, C., and Poshyvanyk, D. (2015). How do developers document database usages in source code? In **Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, 9–13 November 2015, Lincoln, Nebraska, USA**, pages 36–41.

Linares-Vásquez, M., Li, B., Vendome, C., and Poshyvanyk, D. (2016). Documenting database usages and schema constraints in database-centric applications. In **Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, 18–20 July 2016, Saarbrucken, Germany**, pages 270–281. ACM.

Liu, K., Tan, H. B. K., and Chen, X. (2013). Aiding maintenance of database applications through extracting attribute dependency graph. **Journal of Database Management**, 24(1):20–35.

Lopes, S., Petit, J.-M., and Toumani, F. (2002). Discovering interesting inclusion dependencies: application to logical database tuning. **Information Systems**, 27(1):1–19.

Luenberger, D. G. and Ye, Y. (2015). **Linear and Nonlinear Programming**. Springer Publishing Company, Incorporated.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. **The Annals of Mathematical Statistics**, 18(2):50–60.

Markowitz, V. M. and Makowsky, J. A. (1990). Identifying extended entity-relationship object structures in relational schemas. **Transactions on Software Engineering**, 16(8):777–790.

Maule, A., Emmerich, W., and Rosenblum, D. S. (2008). Impact analysis of database schema changes. In **Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, 10–18 May 2008, Leipzig, Germany**, pages 451–460. ACM.

McIntosh, S., Adams, B., and Hassan, A. E. (2012). The evolution of java build systems. **Empirical Software Engineering**, 17(4-5):578–608.

Meurice, L. and Cleve, A. (2014). DAHLIA: A visual analyzer of database schema evolution. In **Proceedings of IEEE CSMR-WCRE 2014 Software Evolution Week, 3–6 February 2014, Antwerp, Belgium**, pages 464–468.

Meurice, L. and Cleve, A. (2016). Dahlia 2.0: A visual analyzer of database usage in dynamic and heterogeneous systems. In **Proceedings of the 4th IEEE Working Conference on Software Visualization, VISSOFT 2016, 3–4 October 2016, Raleigh, North Carolina, USA**, pages 76–80.

Meurice, L. and Cleve, A. (2017). Supporting schema evolution in schema-less nosql data stores. In **Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, 21–24 February 2017, Klagenfurt, Austria**, pages 457–461.

Meurice, L., Goeminne, M., Mens, T., Nagy, C., Decan, A., and Cleve, A. (2016a). **Analysing the Evolution of Database Usage in Data-Intensive Software Systems in Software Technology: 10 Years of Innovation in IEEE Computer**. John Wiley & Sons. to appear.

Meurice, L., Nagy, C., and Cleve, A. (2016b). Detecting and preventing program inconsistencies under database schema evolution. In **Proceedings of the IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, 1–3 August, Vienna, Austria**, pages 262–273.

Meurice, L., Nagy, C., and Cleve, A. (2016c). Static analysis of dynamic database usage in java systems. In **Proceedings of the 28th International Conference on Advanced Information Systems Engineering, CAiSE 2016, 13–17 June 2016, Ljubljana, Slovenia**, volume 9694 of **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, pages 491–506. Springer Verlag.

Meurice, L., Ruiz, F., Weber, J., and Cleve, A. (2014). Establishing referential integrity in legacy information systems - reality bites! In **Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014, September 28 – October 3 2014, Victoria, BC, Canada**, pages 461–465.

Mordal-Manet, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., and Ducasse, S. (2013). Software quality metrics aggregation in industry. **Journal of Software: Evolution and Process**, 25(10):1117–1135.

Nagy, C., Meurice, L., and Cleve, A. (2015). Where was this sql query executed? a static concept location approach. In **Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, 2–6 March 2015, Montreal, QC, Canada**, pages 580–584.

Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., and Ye, Y. (2002). Evolution patterns of open-source software systems and communities. In **Proceedings of the International Workshop on Principles of Software Evolution, IWPSE 2002, 19–20 May 2002, Orlando, FL, USA**, pages 76–85. ACM.

Naur, P. and Randell, B., editors (1969). **Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968, Brussels, Scientific Affairs Division, NATO**.

Navathe, S. B. and Awong, A. M. (1988). Abstracting relational and hierarchical data with a semantic data model. In **Proceedings of the 6th International Conference on Entity-Relationship Approach, ER 1987, 9–11 November 1987, New York, USA**, pages 305–333. North-Holland Publishing Co.

Ngo, M. N. and Tan, H. B. K. (2008). Applying static analysis for automated extraction of database interactions in web applications. **Information and Software Technology**, 50(3):160.

Noughi, N. and Cleve, A. (2015). Conceptual interpretation of sql execution traces for program comprehension. In **Proceedings of the 6th International IEEE Workshop on Program Comprehension through Dynamic Analysis, PCODA 2015, March 2 2015, Montreal, QC, Canada**.

Olivera, H. V., Holanda, M., Guimarâes, V., Hondo, F., and Boaventura, W. (2015). Data modeling for nosql document-oriented databases. In Ventura, J. A. L. and Alatrista-Salas, H., editors, **Proceedings of the 2nd International Symposium on Information Management and Big Data, SIMBig 2015, 2–4 September 2015, Cusco, Peru**, volume 1478 of **CEUR Workshop Proceedings**, pages 129–135. CEUR-WS.org.

Pannurat, N., Kerdprasop, N., and Kerdprasop, K. (2010). Database reverse engineering based on association rule mining. **Computing Research Repository**, abs/1004.3272.

Papastefanatos, G., Anagnostou, F., Vassiliou, Y., and Vassiliadis, P. (2008). Hecataeus: A what-if analysis tool for database schema evolution. In **Proceedings of the 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, 1–4 April 2008, Athens, Greece**, pages 326–328.

Papastefanatos, G., Vassiliadis, P., Simitsis, A., and Vassiliou, Y. (2007). What-if analysis for data warehouse evolution. In Song, I., Eder, J., and Nguyen, T., editors, **Data Warehousing and Knowledge Discovery**, volume 4654 of **LNCS**, pages 23–33. Springer.

Papastefanatos, G., Vassiliadis, P., Simitsis, A., and Vassiliou, Y. (2010). Hecataeus: Regulating schema evolution. In **Proceedings of the 26th IEEE International Conference on Data Engineering, ICDE 2010, 1–6 March 2010, Long Beach, California, USA**, pages 1181–1184.

Petit, J.-M., Kouloumdjian, J., Boulicaut, J.-F., and Toumani, F. (1994). Using queries to improve database reverse engineering. In **Proceedings of the 13th International Conference on the Entity-Relationship Approach, ER 1994, 13–16 December 1994, Manchester, UK**, pages 369–386. Springer-Verlag.

Pfleeger, S. L. (2008). Software metrics: Progress after 25 years? **IEEE Software**, 25(6):32–34.

Phil Factor (2010). https://www.simple-talk.com/blogs/listing-common-sql-code-smells/. Accessed: 2017-04-04.

Premerlani, W. J. and Blaha, M. R. (1994). An approach for reverse engineering of relational databases. **Communications of the ACM**, 37(5):42–ff.

Qiu, D., Li, B., and Su, Z. (2013). An empirical analysis of the co-evolution of schema and code in database applications. In **Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, 18–26 August 2013, Saint Petersburg, Russia**, pages 125–135. ACM.

Queille, J.-P. and Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In **Proceedings of the 5th Colloquium on International Symposium on Programming, 6–8 April 1982, Turin, Italy**, pages 337–351. Springer-Verlag.

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

Ramdoyal, R., Cleve, A., and Hainaut, J.-L. (2010). Reverse engineering user interfaces for interactive database conceptual analysis. In **Proceedings ot the 22nd International Conference on Advanced Information Systems Engineering, CAISE 2010, 7–11 June 2010, Hammamet, Tunisia**, volume 6051 of **LNCS**, pages 332–347. Springer.

Ringlstetter, A., Scherzinger, S., and Bissyandé, T. F. (2016). 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 2016, 16 May 2016, Austin, Texas, USA**, pages 33–36. ACM.

Robles, G., Gonzalez-Barahona, J. M., and Herraiz, I. (2009). Evolution of the core team of developers in libre software projects. In **Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, MSR 2009, 16–17 May 2009, Vancouver, BC, Canada**, pages 167–170.

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. In **Proceedings of the 9th IEEE International Conference on Software Engineering, ICSE 1987, March 30 – April 2 1987, Monterey, California, USA**, pages 328–338.

Ruttan, J. (2008). **The Architecture Of Open Source Applications**, volume II, chapter OSCAR.

Scherzinger, S., Cerqueus, T., and d. Almeida, E. C. (2015). Controvol: A framework for controlled schema evolution in nosql application development. In **Proceedings of the IEEE 31st International Conference on Data Engineering, ICDE 2015, 13–16 April 2015, Seoul, Korea**, pages 1464–1467.

Scherzinger, S., De Almeida Cunha, E., Ickert, F., and Del Fabro Didonet, M. (2013). On the necessity of model checking nosql database schemas when building saas applications. In **Proceedings of the International Workshop on Testing the Cloud, TTC 2013, 15–20 July 2013, Lugano, Switzerland**, pages 1–6. ACM.

Shapiro, S. S. and Wilk, M. B. (1965). An Analysis of Variance Test for Normality (Complete Samples). **Biometrika**, 52(3/4):591–611.

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

Skoulis, I., Vassiliadis, P., and Zarras, A. (2014). Open-source databases: Within, outside, or beyond lehman's laws of software evolution? In **Proceedings of 26th International Conference on Advanced Information Systems Engineering, CAISE 2014, 16–20 June 2014, Thessaloniki, Greece**, volume 8484 of **LNCS**, pages 379–393. Springer.

Stonebraker, M. (2011). Stonebraker on nosql and enterprises. **Communications of the ACM**, 54(8):10–11.

Terwilliger, J. F. et al. (2006). The user interface is the conceptual model. In **Proceedings of the 25th International Conference on Conceptual Modeling, ER 2006, 6–9 November 2006, Tucson, AZ, USA**, volume 4215 of **LNCS**, pages 424–436. Springer.

TIOBE Programming Community Index (2017). http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2017-02-10.

Vasilescu, B., Serebrenik, A., and van den Brand, M. (2011). You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In **Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM 2011, 25–30 September 2011, Williamsburg, VA, USA**, pages 313–322.

Vassiliadis, P., Zarras, A. V., and Skoulis, I. (2015). How is life for a table in an evolving relational schema? birth, death and everything in between. In **Proceedings of**

the 34th International Conference on Conceptual Modeling, ER 2015, 19–22 October 2015, Stockholm, Sweden, pages 453–466.

Vassiliadis, P., Zarras, A. V., and Skoulis, I. (2017). Gravitating to rigidity: Patterns of schema evolution – and its absence – in the lives of tables. **Information Systems**, 63:24 – 46.

Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H., and Yu, J. X. (2010). Matching dependence-related queries in the system dependence graph. In **Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, 20–24 September 2010, Antwerp, Belgium**, pages 457–466.

Wassermann, G., Gould, C., Su, Z., and Devanbu, P. (2007). Static checking of dynamically generated queries in database applications. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, 16(4).

Wei, K., Muthuprasanna, M., and Kothari, S. C. (2006). Preventing sql injection attacks in stored procedures. In **Proceedings of the 17th Australian Software Engineering Conference, ASWEC 2006, 18–21 April 2006, Sydney, Australia**, pages 191–198.

Weiser, M. (1981). Program slicing. In **Proceedings of the 5th IEEE International Conference on Software Engineering, ICSE 1981, 9–12 March 1981, San Diego, California, USA**, pages 439–449.

Weiss, M., Moroiu, G., and Zhao, P., editors (2006). **Evolution of Open Source Communities**. Springer US.

Wermelinger, M., Yu, Y., Lozano, A., and Capiluppi, A. (2011). Assessing architectural evolution: a case study. **Empirical Software Engineering**, 16(5):623–666.

Wettel, R. and Lanza, M. (2008a). Codecity: 3d visualization of large-scale software. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, **Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, 10–18 May 2008, Leipzig, Germany**, pages 921–922. ACM.

Wettel, R. and Lanza, M. (2008b). Visual exploration of large-scale system evolution. In Hassan, A. E., Zaidman, A., and Penta, M. D., editors, **Proceedings of the 15th IEEE Working Conference on Reverse Engineering, WCRE 2008, 13 June 2008, Antwerp, Belgium**, pages 219–228.

Wettel, R., Lanza, M., and Robbes, R. (2011). Software systems as cities: a controlled experiment. In **Proceedings of the 23rd International Conference on Software Engineering, ICSE 2011, 12–19 May 2011, Toronto, Ontario, Canada**, pages 551–560.

Yao, H. and Hamilton, H. J. (2008). Mining functional dependencies from data. **Data Mining and Knowledge Discovery**, 16(2):197–219.

Ye, Y., Nakakoji, K., Yamamoto, Y., and Kishida, K. (2005). **The co-evolution of systems and communities in Free/Open Source Software Development**, pages 59–83. IGI Global.

Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. **IEEE Transactions on Software Engineering**, 30(9):574–586.

Zaidman, A., Rompaey, B. V., van Deursen, A., and Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. **Empirical Software Engineering**, 16(3):325–364.

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