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

Loup Meurice, Mathieu Goeminne, Tom Mens,
Csaba Nagy, Alexandre Decan, and Anthony Cleve

October 26, 2015

Abstract

This chapter presents the research advancements in the field of data-intensive software system evolution, 5 years after the publication of our IEEE Computer column presenting the challenges in this field [1]. We present the state-of-the-art in this research domain, and report on research on the evolution of open source Java projects relying on relational database technologies. We empirically analyse how the use of Java database technologies evolves over time. We report on a coarse-grained source-code analysis carried out over several thousands of Java projects, and complement this by a fine-grained longitudinal analysis of the co-evolution between database schema changes and source code changes within three large Java projects. The presented results are a first step towards a recommendation system supporting developers in writing database-centered code.

1 Introduction

Our August 2010 IEEE Computer column [1] reported on four important challenges that developers of evolving *data-intensive software systems* are confronted with. By *data-intensive* we understand any *software system* (i.e., a collection of programs implementing the business logic) that strongly interacts with a *database* (containing the business data, e.g., customers, invoices, shipments, that form an accurate image of the business). While the software system is implemented in one or more programming languages (e.g., Java), the business data is managed by a (often relational) database management system (DBMS). The database is structured according to a schema that faithfully models the business structure and its rules.

It is widely known that any software system is subject to frequent changes [2]. These changes can have many causes, including requirements changes, technological changes (e.g., new technology or new languages), and structural changes that aim to improve quality (in either the programs or the data). While both the software and database engineering research communities have addressed such evolution problems separately, the challenge is how to cope with the co-evolution of the software system and its data system.

The link between the software part and the data part of a data intensive system is ensured through some API, library or framework that takes care of connecting the program code to the database. In the simplest case, the code will contain embedded database queries (e.g., SQL statements) that will be dynamically built by the application programs and then interpreted by the DBMS. In more complex cases, especially for object-oriented programming languages such as Java, object-relational mappings (ORM) will be provided to translate the concepts used by the program (e.g., classes, methods and attributes) into concepts used by the database (e.g., tables, columns and values).

While an ORM mainly serves to tackle the so-called *object-relational impedance mismatch* [3], this comes at a certain cost. For instance, ORM middleware provides programmers an external, object-oriented view on the physical, relational database schema. Both schemas can evolve asynchronously, each at their own pace, often under the responsibility of independent teams. Severe inconsistencies between system components 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 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. In such a context, co-evolving the database and the program requires to master several different languages, frameworks and APIs.

This chapter reports on the research progress we have recently achieved in this domain, and situate it in the light of the achievements of other researchers. Our research has been focused on Java projects, because Java is one of the most popular object-oriented programming languages today, and because it offers a wide variety of frameworks and APIs for providing an ORM or other means of communicating with a DBMS. To better understand how the use of such technologies within software projects evolves over time, we carried out empirical studies at different levels of granularity. At a coarse-grained level, we analysed and compared the evolution of database technologies used in the source code of thousands of Java projects. At a fine-grained level, we studied the co-evolution between database schema changes and source code changes, focusing on a limited number of Java systems and database technologies.

This chapter is structured as follows. Section 2 presents the state of the art in research on the evolution of data-intensive software systems, and puts our own research into perspective. Section 3 provides an overview of the empirical approach that we have been following to study the co-evolution of Java-based software systems interacting with relational database systems. Section 4 presents some of the findings based on a coarse-grained empirical analysis of several thousands of systems. Section 5 explores three such systems at a fine-grained level of detail, studying the co-evolution between their software and database parts, taking into account the ORM that relates them. Section 6 concludes, and Section 7 presents some open avenues of further research.

2 State of the art

While the literature on database schema evolution is very large [4], few authors have proposed approaches to systematically *observe* how developers cope with database evolution in practice.

Sjoberg [5] presented a study where the schema evolution history of a large-scale medical application is measured and interpreted. Curino *et al.* [6] focused on the structural evolution of the Wikipedia database, with the aim to extract both a micro- and a macro-classification of schema changes. Vassiliadis et al. [7] studied the evolution of individual database tables over time in eight different software systems. They also tried to determine whether Lehman's laws of software evolution hold for evolving database schemas as well [8]. 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.

Several researchers have tried to identify, extract and analyse database *usage* in application programs. The purpose of the proposed approaches ranges from error checking [9, 10, 11], over SQL fault localization [12], to fault diagnosis [13]. More recently, Linares-Vasquez et al. [14] studied how developers *document* database usage in source code. Their results show that a large proportion of database-accessing methods is completely undocumented.

Several researchers have also studied the problem of database schema evolution *in combination* with source code evolution. Maule et al. [15] studied a commercial object-oriented content management system to statically analyze the impact of relational database schema changes on the source code. Chen et al. [16] proposed a static code analysis framework for detecting and fixing performance issues and potential bugs in ORM usage. Their analysis revealed that the modifications made after analysis caused an important improvement of the studied systems' response time. Qiu et al. [17] empirically analyzed the co-evolution of relational database schemas and code in ten open-source database applications from various domains. They studied specific change types inside the database schema and the impact of such changes on PHP code. Karahasanoić [18] 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. However, he focused on object-oriented databases rather than relational databases. Lin et al. [19] 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.

From a less technical point of view, Riaz et al. [20] conducted a survey with software professionals in order to determine the main characteristics that predict maintainability of relational database-driven software applications. It would be interesting to see to which extent these subjective opinions obtained from professionals, correspond to the actual maintainability problems that can be observed by analysing the evolution history of the source code and database schemas directly.

Our own research

Our own research focuses on the empirical analysis of the co-evolution of object-oriented code and relational database schemas, restricted mainly to open source Java systems and ORM technologies, and studying both the technical and social aspects.

In [21], we empirically analyzed the evolution of the usage of SQL, Hibernate and JPA in a large and complex open source information system, called *OSCAR*, that has been implemented in Java. We observed a migration to ORM and persistence technologies offered by Hibernate and JPA, but the practice of using embedded SQL code still remains prevalent today. Contrary to our intuition, we did not find a specialization of developers towards specific database-related activities: the majority of developers appeared to be active in both database-unrelated and database-related activities, and this during the entire considered time period. As a parallel research track we validated on the *OSCAR* system a tool-supported method for analyzing the evolution history of legacy databases [22]. We extracted the logical schema for each system version from the SQL files collected from the versioning system. Then we incrementally built a historical schema which we finally visualized and analyzed further. This analysis focused on the database and did not consider the application code. In [23] we studied both the database schema and the application code in order to identify referential integrity constraints. We demonstrated our approach on the Oscar system by searching foreign key candidates. We analyzed the database schema, the embedded SQL statements in the Java code, and the JPA object model. In [24] we presented a tool-supported technique allowing to locate the source code origin of a given SQL query in hybrid data-intensive systems that rely on JDBC, Hibernate and/or JPA to access their database.

Complementing the research of [21], in [25] we 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 [26]. Using the statistical technique of survival analysis, we explored the survival of the database technologies in the considered projects. In particular, we analysed whether certain technologies co-occur frequently, and whether some technologies get replaced over time by others. We observed that some combinations of database technologies appeared to complement and reinforce one another. We did not observe any evidence of technologies disappearing at the expense of others.

With respect to tool support, we developed DAHLIA, a tool to visually analyze the database schema evolution [27]. DAHLIA provides support for both 2D and 3D visualization. The 2D mode proposes an interactive panel to investigate the database objects (e.g. tables, columns, foreign keys and indexes) of the historical schema and query their respective history. The 3D mode makes use of the city metaphor of CodeCity [28].

3 Analysing the use of ORM technologies in database-driven Java systems

For the current chapter, we limited ourselves to mine historical information from large *open source 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 (August 2015). In addition to this, a large number of technologies and frameworks have been provided to facilitate database access from within Java code.

The choice for open source systems is motivated by the accessibility of source code. The source

code development history for the chosen systems was extracted from the project repositories available through the Git distributed version control system. This enables us to analyze the evolution over time of the project activity and the database usage by the considered systems.

A large-scale empirical analysis, carried out in [25], revealed that a wide range of frameworks and APIs are used by open source Java projects to facilitate relational database access. These technologies operate on different levels of abstraction. For example, as illustrated by Figure 1, a developer can simply choose to embed character strings that represent SQL statements in the source code. The SQL queries are sent to the database through a connector like JDBC, which provides a low-level abstraction of SQL-based database access.

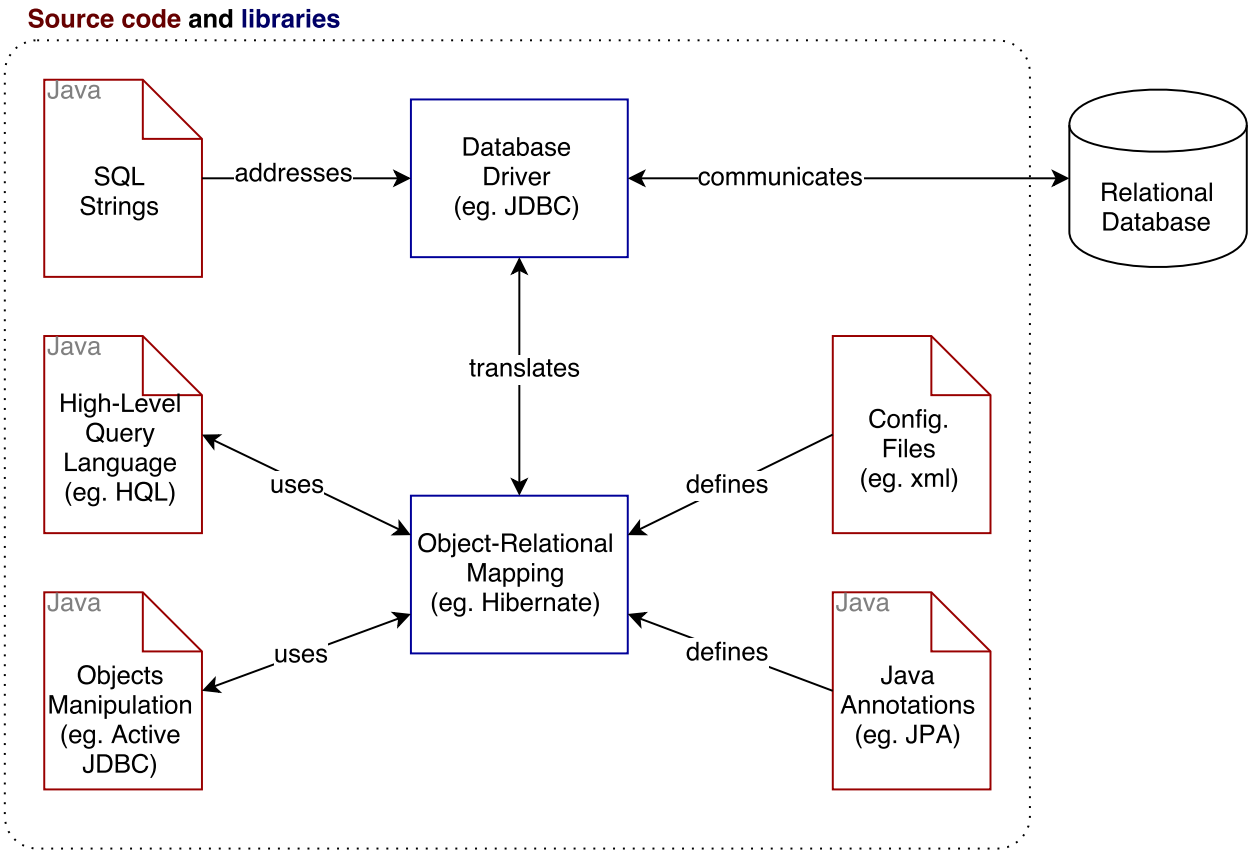


Figure 1: Schematic overview of the interaction between source code and a relational database.

Interaction with the database can also be realized using an ORM library. These libraries offer a higher level of abstraction based on a mapping between Java classes and database tables.

The mapping can take many different forms, as illustrated in Figure 2. The first example shows the use of Hibernate configuration files (i.e., `.hbm.xml`). The second example illustrates the use of Java annotations based on JPA, the standard API in Java for ORM and data persistence management. Such a mapping may allow direct operations on objects, attributes and relationships instead of tables and columns. This is commonly referred as the *active record pattern*.

Some ORM libraries also provide SQL-inspired languages that allow to write SQL-like queries

using the mappings defined before. The third example in Figure 2 illustrates the Hibernate Query Language (HQL), and the fourth example uses the Java Persistence Query Language (JPQL), a platform-independent object-oriented query language which is defined as part of the JPA specification.

Example of using Hibernate configuration files:

```
<hibernate-mapping>
 <class name="Customer" table="AppCustomers">
   <id name="id" type="int" column="id"/>
   <property name="name" column="name" type="String"/>
 </class>
</hibernate-mapping>
```

Example of using JPA annotations:

```
@Entity
@Table(name="AppCustomers")
public class Customer {
   @Id
   private int id;
   String name;
   [...]
}
```

Example of using Hibernate Query Language (HQL):

```
public List<Customer> findAllCustomers(){
   String hql = "select c from Customer c";
   return executeQuery(hql);
}
public List executeQuery(String hql){
   return session.createQuery(hql).list();
}
public Customer findCustomer(Integer id){
   return (Customer) session.get(Customer.class, id);
}
```

Example of direct object manipulation using ActiveJDBC:

```
List<Customer> customers = Customer.where("name = 'John Doe'");
Customer john = customers.get(0);
john.setName("John Smith");
john.saveIt();
```

Figure 2: Four examples of Java ORM usage.

There are also popular web application frameworks with database access APIs among their features. For instance, the *Spring* framework provides interfaces to make it easier to use JPA or to access a database through JDBC. It offers solutions for resource management, data access object (DAO) implementations or transaction handling. Typically, it supports the integration with popular ORM frameworks such as Hibernate.

Co-Evolution

The plethora of different database access technologies available for Java systems, combined with the omnipresence of relational SQL-based database technologies, inspired us to study the problem of co-evolution from two different points of view.

From a coarse-grained, general point of view, we want to understand if and how different database access technologies for Java are used together and how this evolves over time. Do some technologies reenforce one another? Do newer technologies tend to replace older ones? Since the use of database access technologies may differ significantly from one Java system to another, an answer to such questions requires a large-scale longitudinal study over several hundreds or thousands of Java projects in order to come to conclusive, statistically relevant results. This is what we will present in Section 4.

From a fine-grained, system-specific point of view, we also want to determine how database tables and columns evolve over time, and how given database tables and columns are accessed by the programs source code, through which technologies, in which classes and methods, and through which queries. Addressing such questions is highly useful in the context of data-intensive systems maintenance; in particular to achieve a graceful co-evolution of the database and the programs. The answers to those questions being system-specific, they cannot be generalized. Therefore, Section 5 will focus on automated fine-grained analysis of three particular data-intensive systems.

4 Coarse-grained analysis of database technology usage

Despite the fact that database technologies are crucial for connecting the source code to the database, a detailed study of their usage and their evolution is generally neglected in scientific studies. At a coarse-grained level of abstraction, we wish to understand how existing open source Java systems rely on relational database technologies, and how the use of such technologies evolves over time.

By doing so, we aim to provide to developers and project managers a historical overview of database technologies usage that helps them to evaluate the risks and the advantages of using a technology or a combination of technologies. In particular, empirical studies can help to if (and which) database technologies are often replaced and if they can remain used in Java projects for a long time before becoming completed with or substituted by another technology.

Selected Java projects and relational database technologies

In order to carry out such a coarse-grained empirical study, we extracted Java projects from the *Github Java Corpus* proposed by Allamanis and Sutton [26]. In total, we studied 13,307 Java

projects that still had an available Git repository on 24 March 2015. By skimming recent scientific publications, Stack Exchange and blog posts, we identified 26 potential relational database technologies for Java. As a constraint, we imposed that the chosen technologies need to have at least a direct means of accessing a relational database. They also need to be identifiable by static analysis. We determined the presence of each of these technologies in each of the 13,307 projects by analyzing the import statements in Java files, as well as the presence of specific configuration files. For the first commit of each month of each considered Java project, we retraced a historical view of the files that can be related to a particular technology or to a particular framework.

This left us with 4,503 Java projects using at least one of the considered database technologies. Based on this first collection of Java projects and database technologies, we narrowed down our selection to consider only the most popular technologies. We identified four technologies that were each used in at least 5% of all considered Java projects in our collection: JDBC (used in 15.1% of all 4,503 projects), JPA (8.1%), Spring (5.7%), and Hibernate (5.4%). The results are summarized in Table 1.

Of all considered Java projects in our collection, only 2,818 of them used at least one of these four technologies at least once. In the remainder of this section, we will therefore only focus on these four technologies and the 2,818 Java projects that use them.

Table 1: Selected Java database technologies.

Used technology	URL	Occurs if the project contains at least a file	#projects
JDBC	www.oracle.com/technetwork/java/javase/jdbc	importing java.sql	2,008
JPA	www.tutorialspoint.com/jpa	importing javax.persistence or java.persistence, or whose filename is meta-inf/persistence.xml	1,075
Spring	projects.spring.io/spring-framework	importing org.springframework or whose name is spring-config.xml or beans.xml	759
Hibernate	hibernate.org	importing org.hibernate or whose filename ends with .hbm.xml	718

Table 2 presents some size and duration metrics over these 2,818 Java projects. We observe that the distribution of metrics values is highly skewed, suggesting evidence of the Pareto principle [29].

Table 2: Characteristics of the 2,818 considered Java projects.

	mean	standard deviation	minimum	median	maximum
duration (in days) until last considered version	950.8	999.3	0	701	6,728
number of commits	1245.1	5781.6	1	123.5	174,618
number of distinct contributors	12.1	29.6	1	4	1,091
number of files in latest version	1001.2	3384.1	1	195	103,493

Evolution of database technology usage

Since our goal is to study the evolution of database technology usage over time in Java projects, Figure 3 visualizes the evolution over time of the occurrence of the retained technologies in the considered projects. For each technology $\mathsf{T} \in \{\mathsf{JDBC}, \mathsf{Hibernate}, \mathsf{JPA}, \mathsf{Spring}\}$, the y-axis shows the fraction

$$\frac{\text{number of projects using technology } \mathsf{T}}{\text{number of projects using any of the 4 considered technologies}}$$

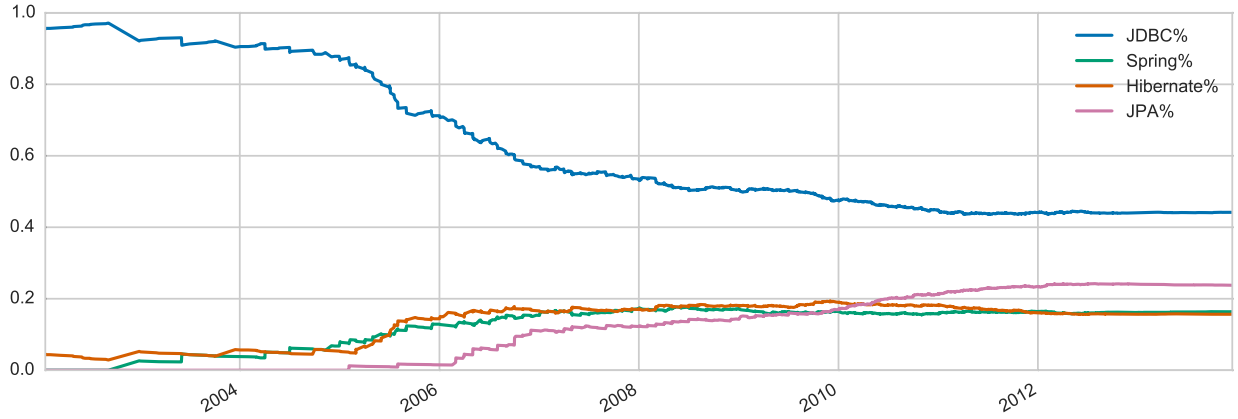


Figure 3: Evolution of the share of each technology $\mathsf{T} \in \{\mathsf{JDBC}, \mathsf{Hibernate}, \mathsf{JPA}, \mathsf{Spring}\}$.

Intuitively, Figure 3 reveals the relative importance (in terms of number of projects) of one technology over another one. (The sum of shares is sometimes greater than 1 because some projects use multiple technologies simultaneously.)

We observe that **JDBC was and remains the most frequent technology**, which is not surprising since it provides the basic means for connecting to a database and handling SQL queries. Nevertheless, **the share of JDBC appears to be decreasing since 2008**, which coincides with the emergence of JPA. We also observe that Hibernate and Spring obtained their maximum before 2010 and since then their shares are slowly decreasing. We hypothesize that this is due to JPA becoming more popular and partially overtaking the other technologies. Indeed, of the four considered technologies, **only JPA's share continues to grow over the entire considered timeframe**.

Co-occurrence of database technologies

Considering the four selected relational database technologies, which combinations of them frequently co-occur in our selection of Java projects? Such information would reveal which technologies are complementary, and which technologies are used as supporting technologies of other ones.

Let's consider the technologies appearing at least once in the entire lifetime of each considered project. Figure 4 shows the number of projects in which a combination of technologies has been detected. A first observation is that **JDBC is the sole technology in 62% of all projects** (1,239

out of 2,008). A possible interpretation is that, in many cases, the services provided by JDBC are considered as sufficient by the developers, and the advantages provided by more advanced technologies do not justify the *cost* of their introduction. On the other hand, we observe that **a large proportion of projects that make use of another technology also make use of JDBC**. For instance, Hibernate is used alone in only 8% of all projects (60/718), and a majority (62%) of all projects that make use of Hibernate also make use of JDBC (but not necessarily at the same time in their history).

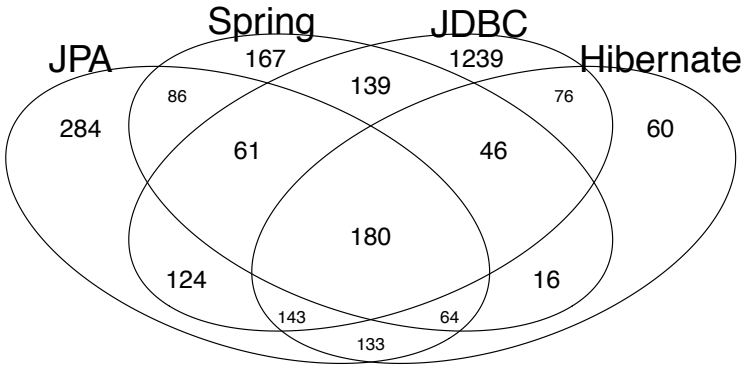


Figure 4: Number of projects for each combination of database technologies.

Something similar can be observed for JDBC and JPA. JPA occurs in isolation in 26% of all projects, while almost half (47%) of all projects that make use of JPA also make use of JDBC. Considering the four selected technologies, **38% of all projects used at least 2 technologies over their observed lifetime**.

These high numbers could be due to the fact that some technologies are used as supporting or complementary technologies for others. More specifically, the low level JDBC is probably punctually used to express complex queries that cannot be expressed –or are difficult to express– with higher-level technologies. However, the reason why 180 of the 2,818 considered projects have used all of the 4 technologies simultaneously remains unknown.

While Figure 4 shows that most projects use several database technologies over their lifetime, it does not provide information about their actual co-occurrences. We therefore compared, for each project, the overall number of detected technologies through the entire lifetime and the maximum number of simultaneously present (i.e., *co-occurring*) technologies. Table 3 presents the detected co-occurrences of technologies. We observed that **in more than 80% of all cases, different database technologies tend to co-occur together in a project**. This represents 887 co-occurrences for 1,068 projects with at least two technologies in their whole lifetime.

While the four technologies present comparable numbers of projects in which they co-occur, these numbers are proportionally very different. For example, **while only 37% of the occurrences of JDBC coincide with another technology, 91% of the occurrences of Hibernate coincide with another technology**. If we zoom in closer, **we observe a co-occurrence with JPA for 77% of all projects in which Hibernate occurs**. This is the highest observed percentage of co-occurrences. On the opposite side of the spectrum, we find that only 43% of all projects in which Hibernate occurs, co-occur with Spring. This is the lowest observed percentage of co-

Table 3: Number of projects in which a technology was used (column 2); absolute and relative number of projects in with the technology co-occurred with one of the 3 other technologies (column 3); absolute and relative number of projects in which a specific pair of database technologies co-occurs (columns 4-7).

			Co-occurrences with...			
Technology	# projects	Co-occurrences with any of the 3 other technologies	JDBC	Spring	Hibernate	JPA
JDBC	2008	743 (37%)	\	363 (49%)	396 (53%)	454 (61%)
Spring	759	582 (77%)	363 (62%)	\	278 (48%)	358 (62%)
Hibernate	718	652 (**91%**)	396 (61%)	278 (43%)	\	502 (**77%**)
JPA	1075	771 (72%)	454 (59%)	358 (46%)	502 (65%)	\

occurrences. Of all considered pairs of technologies, **Spring and Hibernate are used together the least frequently**.

Introduction and disappearance of database technologies

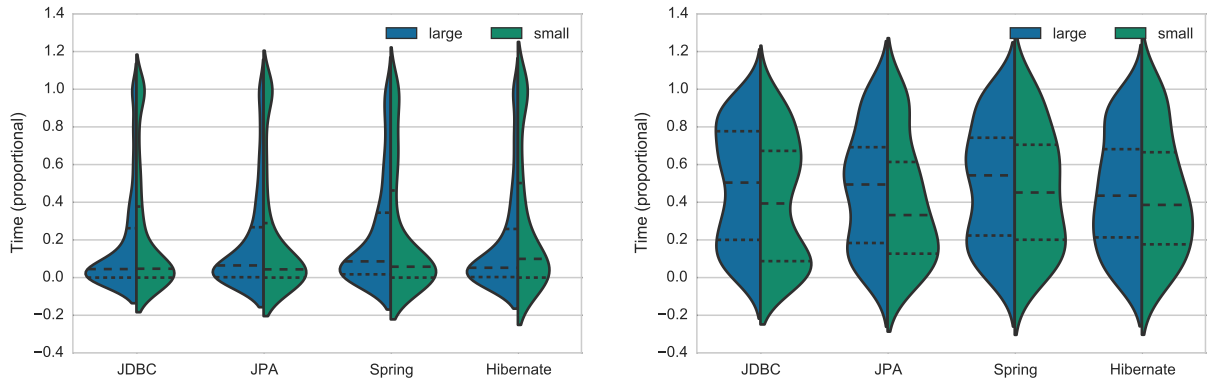
Introducing a new database technology in a software project comes at a certain cost. A common policy is therefore to introduce a new technology only if the expected benefits outweigh the expected cost. Examples of such benefits are more efficient services, increased modularity, a simpler implementation or maintenance, etc.

For each project, we analyzed at what moment in the project's lifetime each occurring database technology got introduced or disappeared. We observed that the answer was strongly related to the duration and size of the considered projects. To take into account the effect of project duration, we normalized the lifetime of each project into a range of values between 0 (the start of the project) and 1 (the last considered commit). To study the effect of project size, we split our project corpus in two equally-sized subsets containing respectively the small and the large projects. Project size was measured as number of files in the latest revision (see Table 2). All small projects contained less than 195 files.

Figure 5 shows the relative moment in the project history where each of the considered technologies have been introduced (Figure 5a) or removed (Figure 5b). Projects in which the considered technology did not disappear before the last observable commit have been disregarded. For both small and large projects, **over 50% of the introductions of a technology are done very early in the projects' life** (in the first 15% of their lifetime). This is what one would expect, since a database (and by extension, a database access technology) is usually part of a project specification since its beginning.

In Figure 5, the distribution of the moment of technology removal is much less skewed than the distribution of the moment of technology introduction. We do not observe particular difference across technologies. Overall, **technologies tend to disappear faster in small than in large projects**, especially for JDBC and JPA.

As many projects use multiple technologies, either simultaneously or one after the other, it is useful to study how the introduction of a new technology can affect the presence of a previous one. We used the statistical technique of *survival analysis* to study this question. This technique creates



(a) introduction of a new technology

(b) disappearance of a technology

Figure 5: Relative time of introduction and disappearance of database technologies. Horizontal dashed lines represent the quartiles of the distribution.

a model estimating the *survival rate* of a population over time, considering the fact that some elements of the population may leave the study, and for some other elements the event of interest does not occur during the observation period. In our case, the observed event is the introduction of a second technology after the introduction of a previously occurring one.

Figure 6 presents the results of the survival analysis, distinguishing the small projects (dashed lines) from the large ones (straight lines). The survival rates of technologies in large projects are significantly lower than for small projects, implying that **new technologies are introduced more often and more quickly in large projects than in small ones**. The survival rates are also significantly higher for JDBC than for the other technologies, both for small and large projects. This indicates that **JPA, Spring and Hibernate rarely succeed JDBC and, if they do, it happens later and more uniformly**. Among the possible explanations, JDBC may be sufficient to satisfy the initial requirements of most projects, while new technologies are only introduced as these requirements change and grow over time.

Figure 6 also reveals that Hibernate has a much lower survival rate than the other technologies. Both small and large **projects tend to complete or to replace Hibernate more often than any other technology**. Finally, we observe that during the first 10% of the projects' lifetime, the survival rates of Hibernate decrease by 15% (resp. 40%) which represents a more important decrease than for the other technologies, which means that **Hibernate is usually quickly followed or complemented by another technology**.

Discussion

We collected and analyzed data for more than 13K Java projects. Of these, 4,503 projects used at least one out of 26 identified Java relational database technologies. Among these technologies, JDBC, JPA, Hibernate and Spring were the most widely used, covering 2,818 of all Java projects. We therefore analyzed the evolution and co-occurrences of those four technologies in order to get a high-level view of their usage in Java projects. As projects often make use of these technolo-

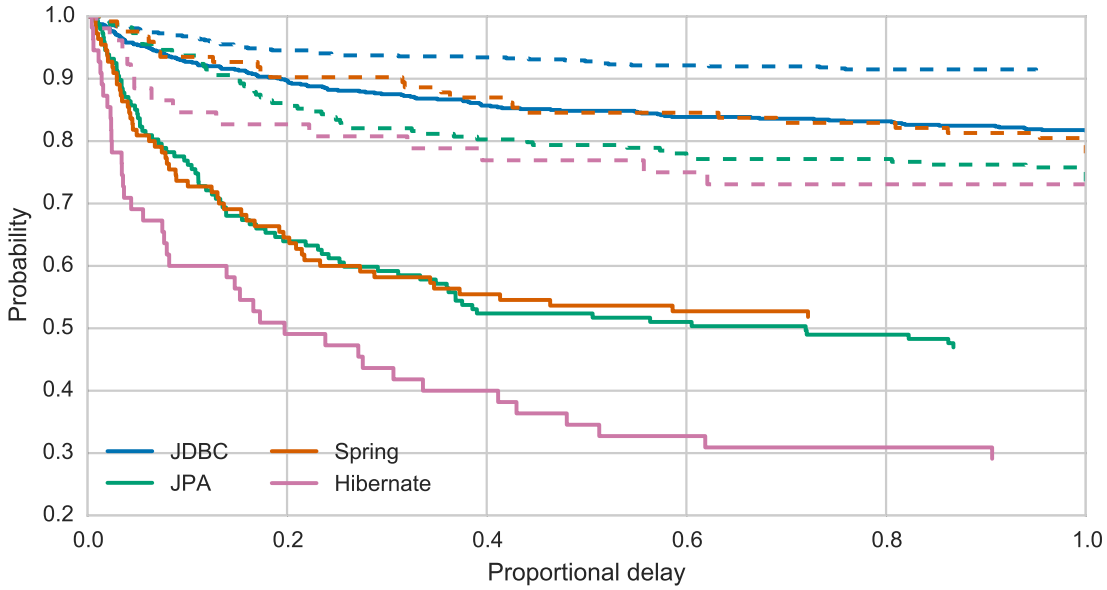


Figure 6: Probability that a technology remains the last introduced technology over time.

gies either simultaneously or one after the other, we deepened our analyses to identify the most frequently co-occurring pairs of technologies, and to determine how fast a technology tends to be replaced by or completed by another one. This coarse-grained analysis allowed us to observe some interesting global trends concerning the four considered technologies (highlighted in **boldface** in the previous subsections).

While a coarse-grained analysis may help to identify such global trends, a more fine-grained analysis is required to target more specific research questions. How and why do developers decide to introduce new database technologies to complement existing ones? How and where are particular technologies used in the source code? Are the same database schema elements covered by different technologies? Do particular code files involve different technologies?

In addition to this, a fine-grained analysis may help to reveal whether the use of particular technologies is in line with the evolution of the database schema. Are some parts of the code out-of-date with respect to the database schema? Is it easier to detect and address these inconsistencies with some of the technologies? Which typical workflows can we observe in the usage of those technologies? Can we quantify and compare the impact on the code of a database change? These are the types of questions that will be targeted in Section 5.

5 Fine-grained analysis of database technology usage

5.1 Analysis background

In our fine-grained analysis, we investigate in the source code and the database schema how some measurable characteristics of database usage in data-intensive systems evolve over time. Our ob-

jective 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.

We studied the evolution of three large open-source Java systems (*OSCAR*, *OpenMRS* and *Broadleaf Commerce*) that have been developed for more than seven years and rely on the database access technologies which we studied in the coarse-grained analysis of Section 4. Two of these systems are popular, Electronic Medical Record (EMR) systems and the third one is an e-commerce framework. All three use a relational database for data persistence and deal with a large amount of data. Hence, they are good representatives of data-intensive software systems. Table 4 presents an overview of the main characteristics of the selected software systems.

Table 4: Main characteristics of the selected systems.

System	Description	KLOC	Start Date
OSCAR	EMR system	$>$ 2,000	11/2002
OpenMRS	EMR system	$>$ 300	05/2006
Broadleaf	E-commerce framework	$>$ 250	12/2008

OSCAR (oscar-emr.com) is an open-source ERM information system that is widely used in the healthcare industry in Canada. Its primary purpose is to maintain electronic patient records and interfaces of a variety of other information systems used in the healthcare industry. *OSCAR* has been developed since 2002. The source code comprises approximately two million lines of code. *OSCAR* combines different ways to access the database because of the constant and ongoing evolution history of the product: the developers originally used JDBC, then Hibernate, and more recently JPA. We empirically confirmed these findings in [21].

OpenMRS (openmrs.org) is a collaborative open-source project to develop software to support the delivery of healthcare in developing countries (mainly in Africa). It was conceived as a general-purpose EMR system that could support the full range of medical treatments. It has been developed since 2006. *OpenMRS* uses a MySQL database accessed via Hibernate and dynamic SQL (JDBC).

Broadleaf (www.broadleafcommerce.org) is an open-source, e-commerce framework written entirely in Java on top of the Spring framework. It facilitates the development of enterprise-class, commerce-driven sites by providing a robust data model, services, and specialized tooling that take care of most of the 'heavy lifting' work. *Broadleaf* has been developed since 2008. It uses a relational database accessed by the source code via JPA.

For all the systems, we performed a static analysis on the source code of selected revisions from the version control systems. Firstly, we picked the initial commits and then we went on through the next revisions and selected those which were at least 15 days from the last selected revision and contained at least 500 modified lines. As a result, we have a snapshot of the state of each system in every 2-3 weeks of its development (the extraction and historization phases are respectively detailed in [24] and [27]). 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 7).

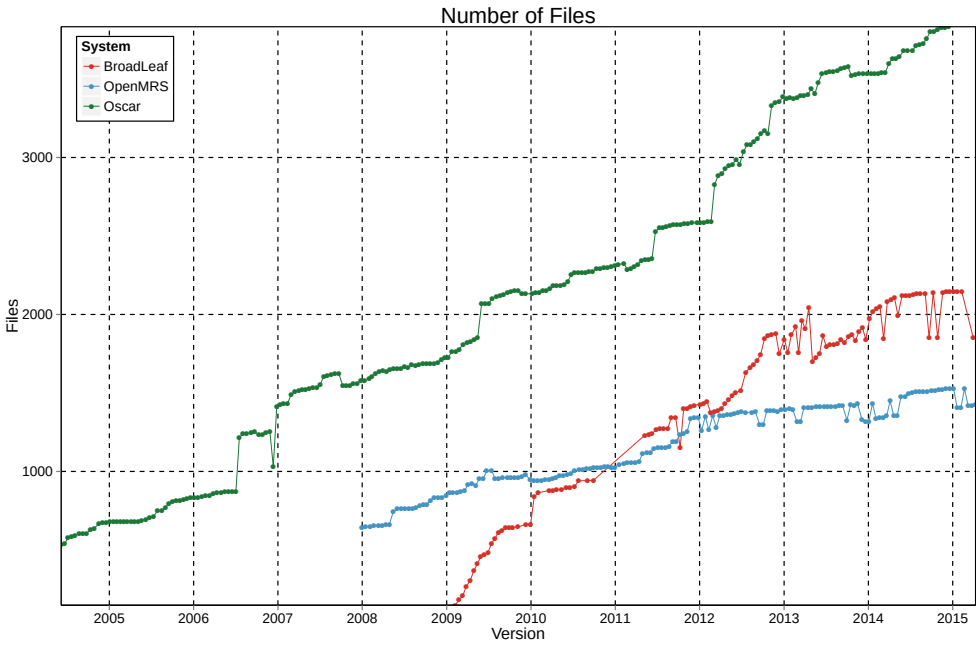


Figure 7: Evolution over time of the number of files in each system.

5.2 Conceptual Schema

We developed static analyzers to compute size metrics of the source code and the database schema. We also identified the database access points in the source code where the database is accessed through JDBC, Hibernate or JPA. These access points could be simple CRUD (Create, Read, Update or Delete) operations, or locations where the system queries the database with native SQL, HQL or JPQL queries.

We used the gathered information to build a generic conceptual schema of all the different artifacts that can be analyzed historically in a given data-intensive system. This schema is presented in Figure 8. The central element is Version, which assigns a unique version identifier to the other artifacts, in order to be able to analyze the evolution of each program component and data component of the system over time. The green elements in the figure represent the source code components (e.g.,File, Class, Method and Attribute (each of them 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). The red elements represent the database components (e.g.,Table and Column). The blue elements represent the DatabaseAccess (e.g., Query and CRUDOperation). Such database accesses appear in the program code as MethodCall to particular methods (at a particular position in the code) that are part of the API, library of framework that takes care of the database access. Database queries (e.g., SQL queries) may be embedded in the code and provide a direct access to some DatabaseObject. Alternatively, the database access may take place through some ORM Mapping (shown by the grey elements in the figure).

Let us illustrate the use of our conceptual schema. Figure 2 shows a sample of Java code using Hibernate for accessing the database in a given system version. In that piece of code, in particular

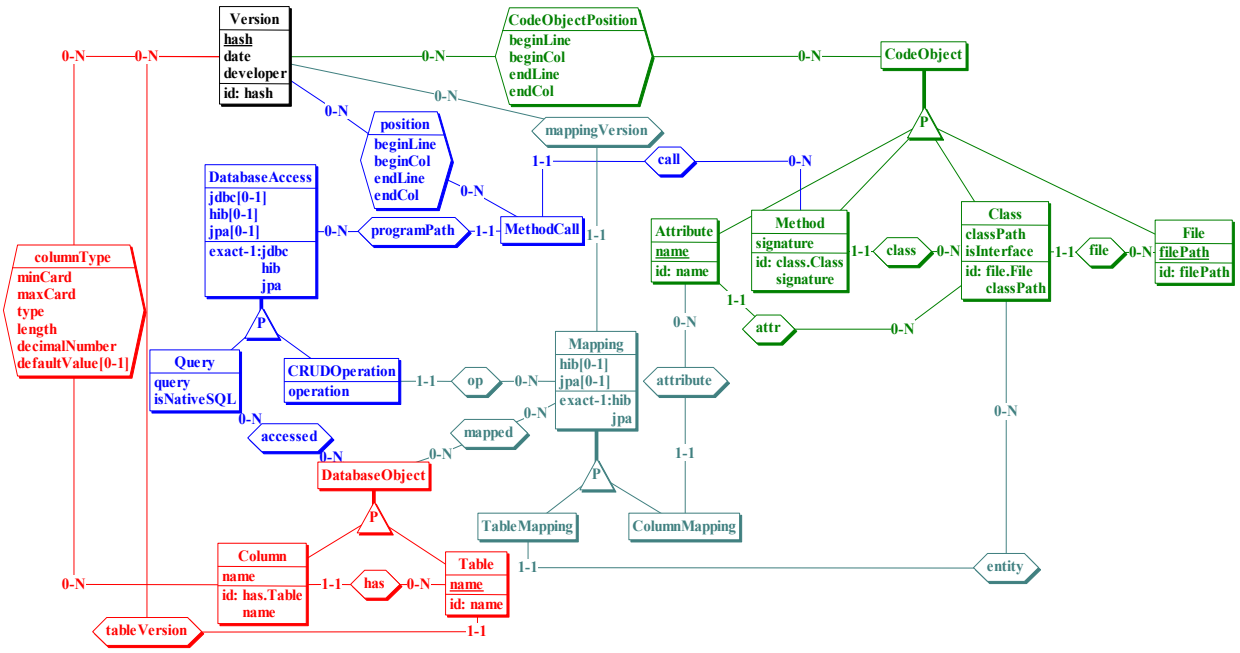


Figure 8: Conceptual schema of the considered DISS artifacts.

in the third example, one easily detects the presence of two database accesses: line 9 retrieves the customer corresponding to a given id, whereas line 3 executes an HQL query selecting all the customers recorded in the system. The first access makes use of `Customer`, a mapped entity Class located in File `/src/pojo/Customer.java`, to query the database. `Customer` is mapped (Mapping) to the `AppCustomers` Table. Line 9 is a Hibernate DatabaseAccess and more precisely a CRUDOperation (of type Read). The ProgramPath of the read access has a length of 1 and is a MethodCall to `findCustomer` Method at line 9 (position). The second database access is an HQL Query accessing the `AppCustomers` Table and has a ProgramPath of length 2: a MethodCall to `findAllCustomers` Method at line 3 and a second to `executeQuery` Method at line 6.

5.3 Metrics

By exploiting this conceptual schema, we studied and measured some characteristics of the three systems at different levels. For measuring those characteristics, we successively analyzed each selected version in order to observe how the systems evolve over time. We respectively selected 242, 164 and 118 versions for *OSCAR*, *OpenMRS* and *Broadleaf*. We now present the different metrics we used to measure the characteristics of each system at the code and database schema levels as well as metrics pertaining to the co-evolution of both.

Each of the three considered systems appears to have its own specific database schema growth trend. Figure 9 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*

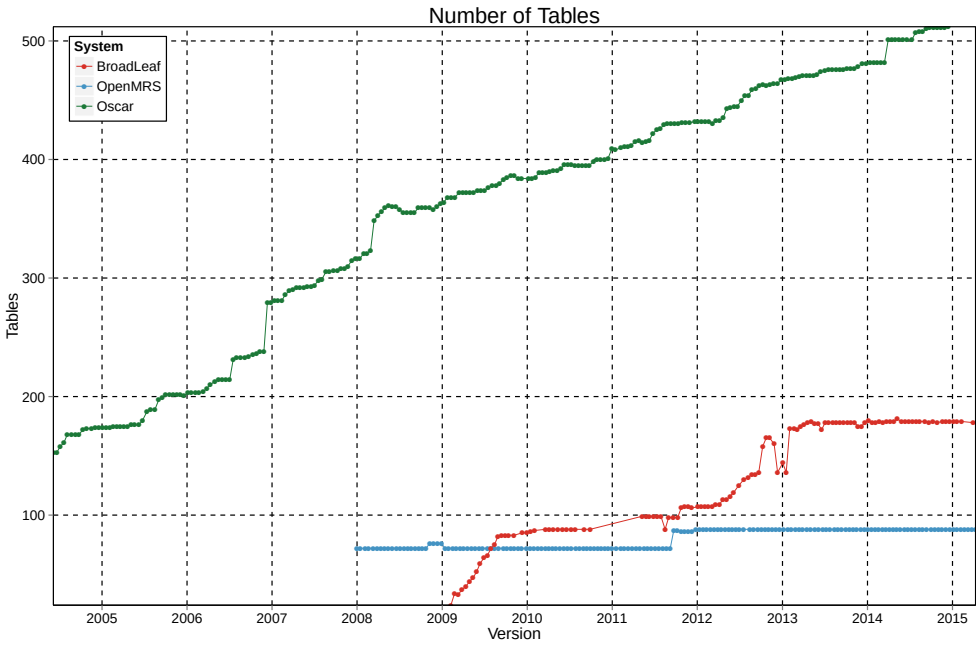


Figure 9: Evolution of the number of tables over time.

growth. Table 5 shows the number of database changes occurred in the life of each system. 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 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.

Table 5: Number of database schema changes in the systems' life.

System	#Added Tables	#Added Columns	#Deleted Tables	#Deleted Columns
Oscar	391	12597	32	2442
OpenMRS	27	438	11	100
Broadleaf	240	1425	86	584

In order to study the co-evolution between the source code and the database schema, we focused on three artifacts of our conceptual schema: (1) the tables that are accessed and the way to access them (Figure 10); (2) the locations of code and files accessing the database (respectively Figure 11 and Figure 12); and (3) the distribution of mappings across ORM technologies (Figure 13).

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 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 Discussion

With our static analysis and the exploitation of our conceptual schema, we studied 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.

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.*" [30]

Compared to *OSCAR*, *OpenMRS* is a less prone to changes. Its code has increasingly evolved over time but there were 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.

Concerning *Broadleaf*, with further analysis of our measurements, we found that the database objects are rarely removed; almost all the removed tables are actually involved in a renaming process. However, as SQL migration scripts are not provided for *Broadleaf*, identifying table renamings is not an easy and direct process. 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 fewer effort since the lines of code that access tables are usually regrouped into a single file.

The question of "what is the required effort to maintain/evolve a given data-intensive software system" remains to be studied in our future work.

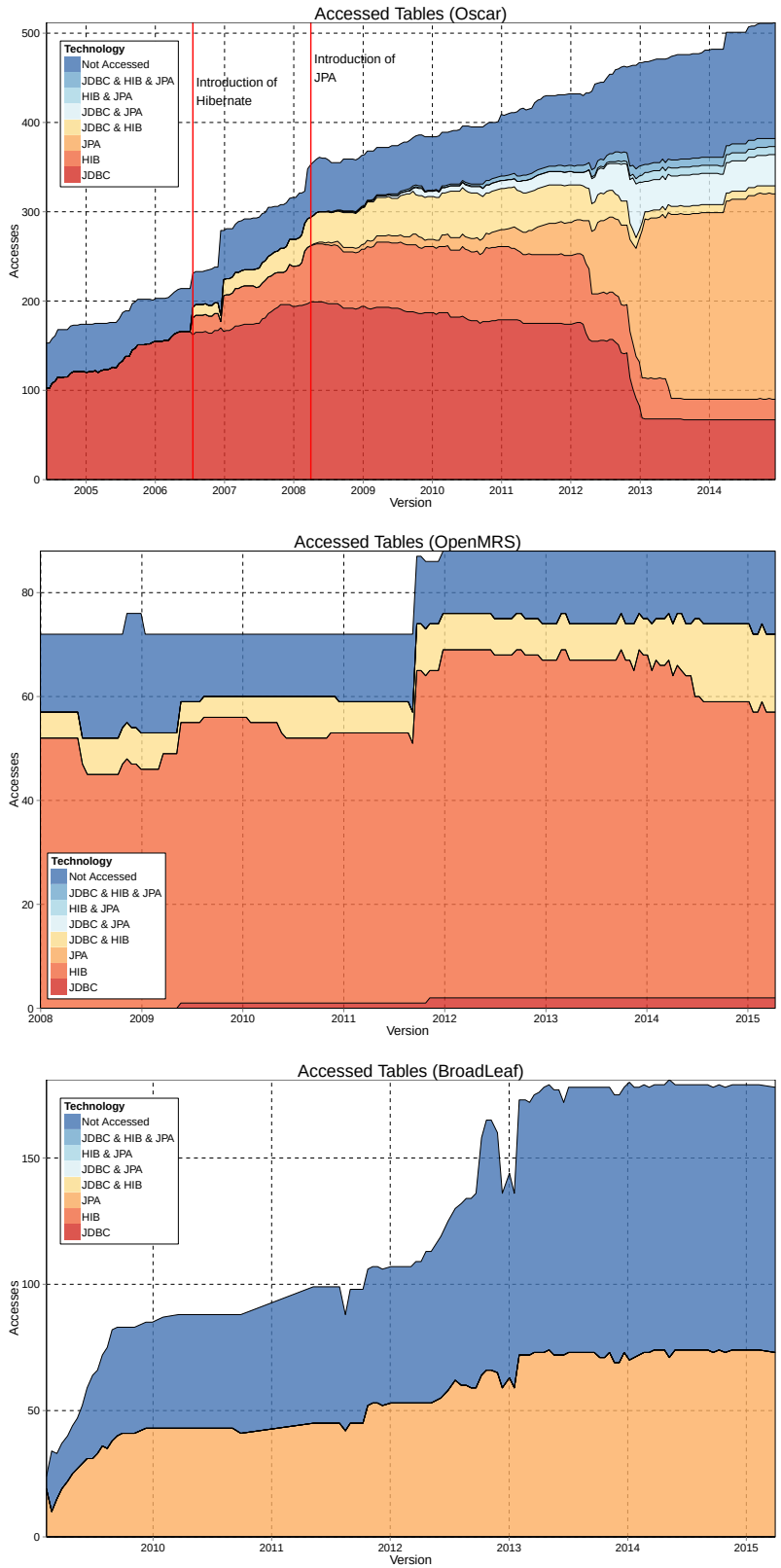


Figure 10: Distribution of the accessed tables across the technologies.

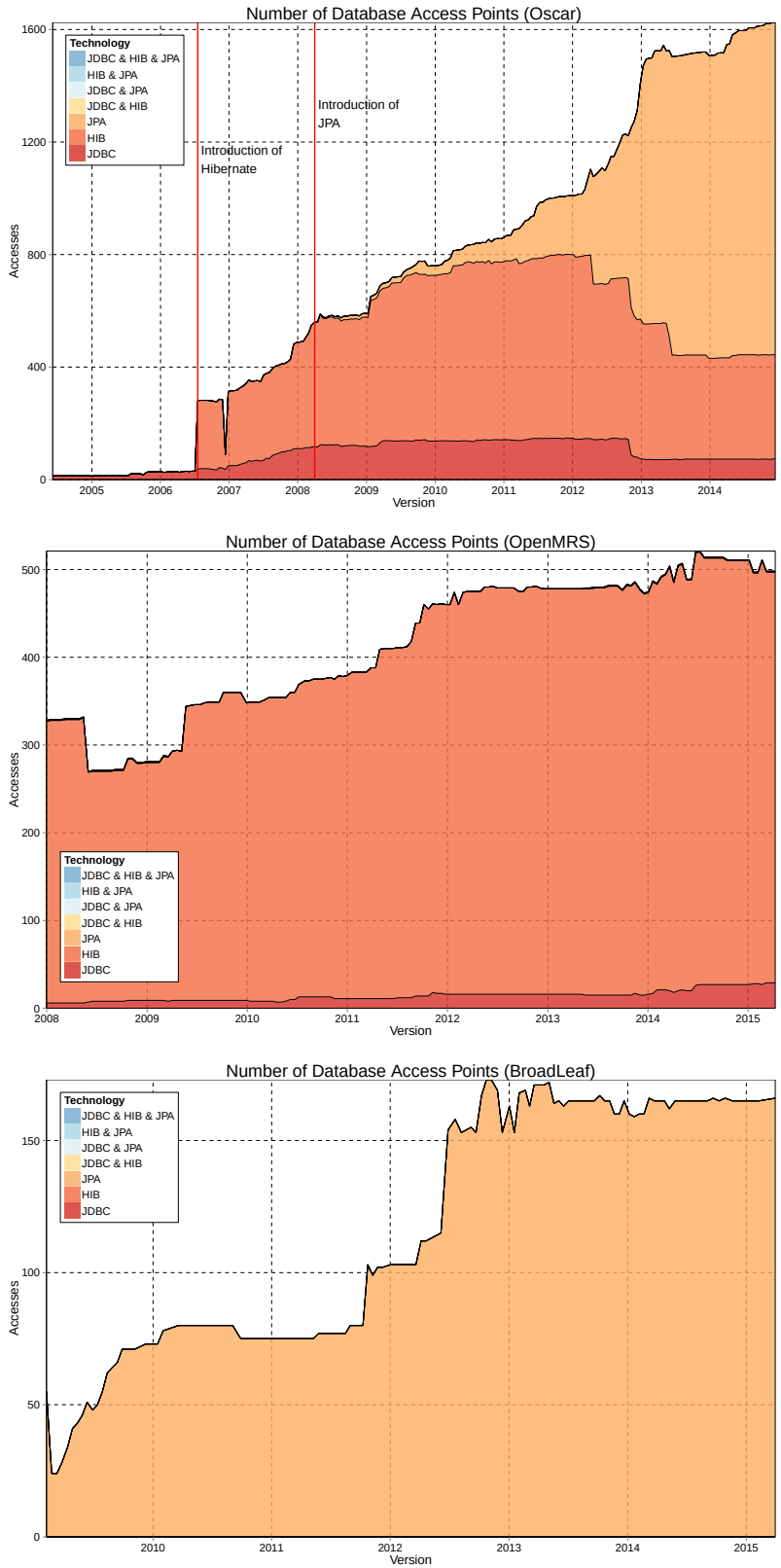


Figure 11: Database access point distribution.

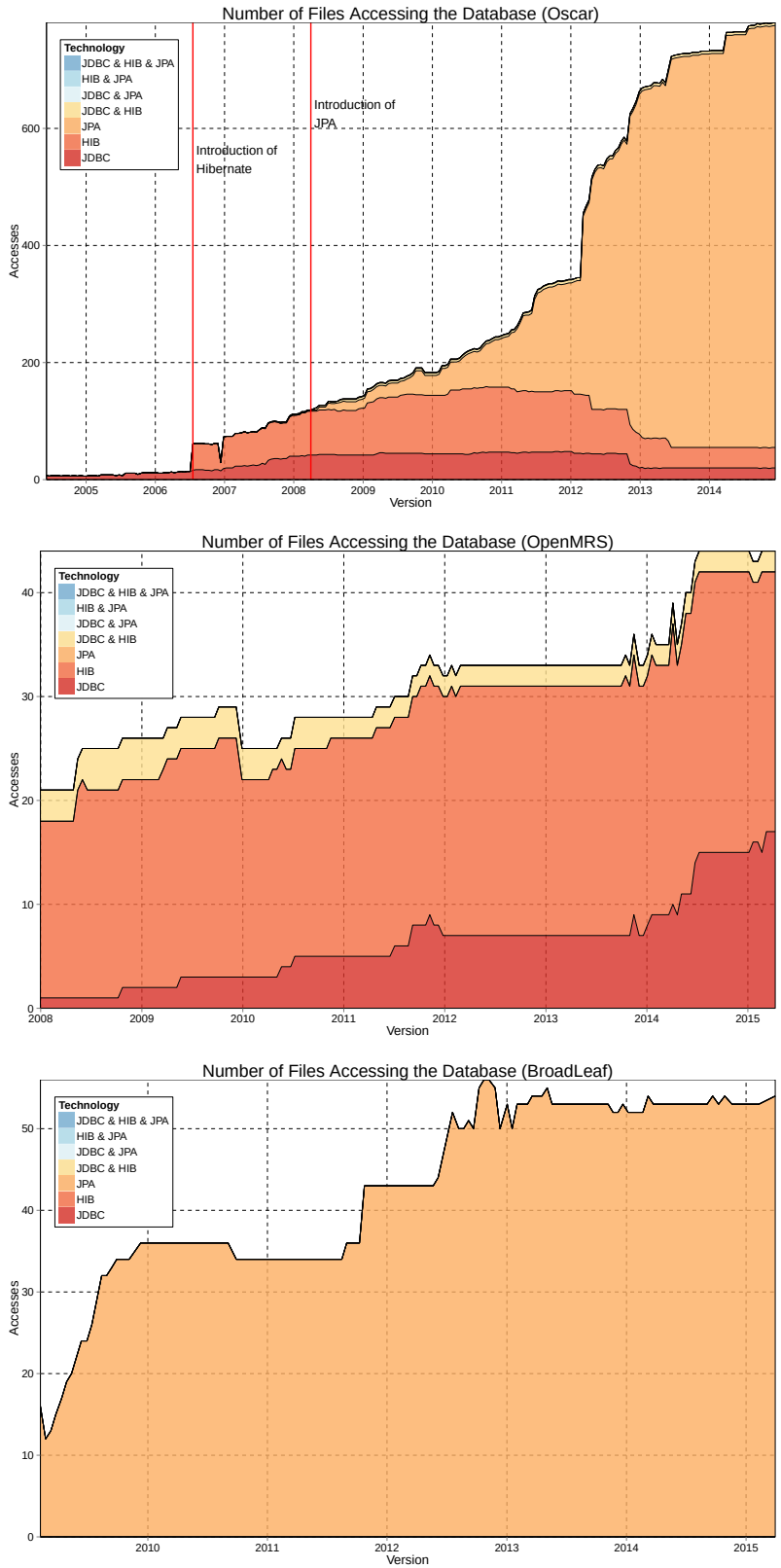


Figure 12: The distribution of the files accessing the database.

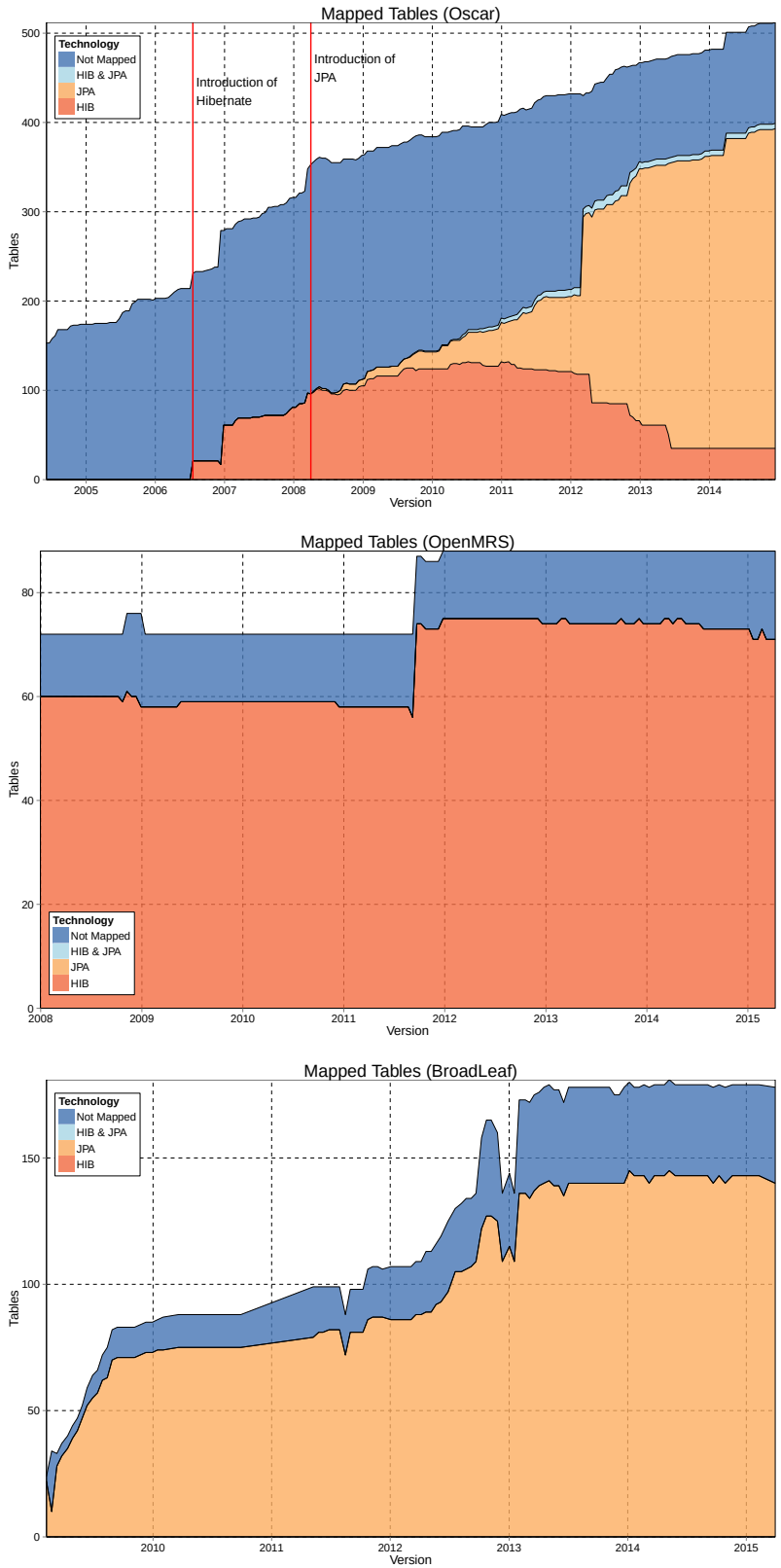


Figure 13: Distribution of the mapped tables across the technologies.

6 Conclusion

Little is known about how, in a data-intensive software system, the software part of the system (i.e., the source code) co-evolves with the data part (e.g., the relational database, represented by a database schema). Empirical software engineering research has mainly focused on studying the evolution of the software system, while ignoring its connected data system.

Even less is known about how this co-evolution is affected by the different database manipulation technologies used to access the database from the source code. In the case of Java software, for example, a wide variety of technologies are used (including JDBC, Hibernate, Spring and JPA).

We carried out a coarse-grained empirical study on several thousands of open source Java software systems to gain insight in how they use database technologies, and how this evolves over time. While low-level technologies like JDBC continue to remain widely used, higher-level ORMs and persistence APIs like JPA are becoming increasingly more popular over time. In addition, we observed that close to 40% of database-driven Java systems have used at least two different technologies over their observed lifetime. For most of them, these different technologies were used simultaneously. We also observed that new technologies are introduced faster in (and disappear slower from) large systems than in small ones. Some technologies, like Hibernate, tend to be replaced faster than other technologies.

We complemented the coarse-grained evolutionary analysis by a fine-grained one in which we narrowed down on three data-intensive open source Java systems. For these systems, we jointly analyzed the changes in the database schema and the related changes in the source code by focusing on the database access locations. We observed, among others, that the very same 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 to *evolve* 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.

We are convinced that the work presented in this chapter can lead to actionable results. They may serve, for instance, as a sound basis to build recommendation systems. Such systems could suggest which developer(s) to contact and what to do if certain changes should be done in some database schema elements or in some program code accessing the database. They could also help development teams to estimate the effort needed to achieve given (co-)evolution tasks, taking into account the particular technology (or technologies) being used.

7 Future Work

This chapter only focused on the technical aspects of how to relate the software and the data parts of a database-driven software system, and how to study their co-evolution over time, taking into account the particular database technologies being used. It is, however, equally important to study the *social aspects* of such systems in order to address a wider range of relevant questions. Are separate persons or teams responsible for managing the program code, the database mapping code, the database? How do developers divide their effort across these different activities? If different technologies are being used, are they used by separate groups of developers? How does all of this change over time? We started to address these questions in [21], but clearly more empirical research is required.

The research results presented here were only focused on Java systems, so similar studies for other programming languages would be needed. Also, the focus was on relational (SQL-based) databases, while many contemporary software systems are relying on NoSQL (i.e., typically non-relational) database management systems.

The empirical analysis that was carried out could be refined in many ways. By using dynamic program analysis as opposed to static programming analysis, at the expense of requiring more data and processing power. By using external data sources such as mailing lists, bug trackers, and Q&A websites (e.g., Stack Overflow) for measuring other aspects such as software quality (e.g., in terms of reported/resolved issues in the bug tracker), developer collaboration (through mailing list communication), user satisfaction (using information from bug tracker and mailing list, combined with sentiment analysis). As a first step in this direction, the use of Stack Overflow has been explored in [31], where it was used to identify error-prone patterns in SQL queries, which is a first step towards a recommendation system supporting developers in writing database-centered code.

Acknowledgments

This research was carried out by the University of Mons and the University of Namur, in the context of a joint research project T.0022.13 “Data-Intensive Software System Evolution” financed by the F.R.S.-FNRS, Belgium.

References

[1] A. Cleve, T. Mens, and J.-L. Hainaut, “Data-intensive system evolution,” *Computer*, vol. 43, no. 8, pp. 110–112, Aug. 2010.

[2] M. M. Lehman, “Laws of software evolution revisited,” in *European Workshop on Software Process Technology (EWSPT)*, 1996, pp. 108–124.

[3] C. Ireland, D. Bowers, M. Newton, and K. Waugh, “A classification of object-relational impedance mismatch,” in *Int’l Conf. Advances in Databases, Knowledge, and Data Applications (DBKDA)*, 2009, pp. 36–43.

[4] E. Rahm and P. A. Bernstein, "An online bibliography on schema evolution," *SIGMOD Rec.*, vol. 35, no. 4, pp. 30–31, Dec. 2006.

[5] D. Sjøberg, "Quantifying schema evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35 – 44, 1993.

[6] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema evolution in wikipedia - toward a web information system benchmark," in *Int'l Conf. Enterprise Information Systems (ICEIS)*, 2008, pp. 323–332.

[7] P. Vassiliadis, A. V. Zarras, and I. Skoulis, "How is life for a table in an evolving relational schema? Birth, death and everything in between," in *Int'l Conf. Conceptual Modeling (ER)*, 2015, pp. 453–466.

[8] I. Skoulis, P. Vassiliadis, and A. Zarras, "Open-source databases: Within, outside, or beyond lehman's laws of software evolution?" in *Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, 2014, pp. 379–393.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Int'l Conf. Static Analysis (SAS)*, 2003, pp. 1–18.

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

[11] M. Sonoda, T. Matsuda, D. Koizumi, and S. Hirasawa, "On automatic detection of SQL injection attacks by the feature extraction of the single character," in *Int'l Conf. Security of Information and Networks (SIN)*, 2011, pp. 81–86.

[12] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL faults in database applications," in *Int'l Conf. Automated Software Engineering (ASE)*, 2011, pp. 213–222.

[13] M. A. Javid and S. M. Embury, "Diagnosing faults in embedded queries in database applications," in *EDBT/ICDT'12 Workshops*, 2012, pp. 239–244.

[14] M. Linares-Vasquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?" in *Int'l Conf. Automated Software Engineering (ASE)*, 2015.

[15] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *Int'l Conf. Software Engineering (ICSE)*, 2008, pp. 451–460.

[16] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Int'l Conf. Software Engineering (ICSE)*, 2014, pp. 1001–1012.

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

[18] A. Karahasanović, "Supporting application consistency in evolving object-oriented systems by impact analysis and visualisation," Ph.D. dissertation, University of Oslo, 2002.

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

[20] M. Riaz, E. Tempero, M. Sulayman, and E. Mendes, "Maintainability predictors for relational database-driven software applications: Extended results from a survey," *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)*, vol. 23, no. 4, pp. 507–522, 2013.

[21] M. Goeminne, A. Decan, and T. Mens, "Co-evolving code-related and database-related changes in a data-intensive software system," in *CSMR-WCRE Software Evolution Week*, 2014, pp. 353–357.

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

[23] L. Meurice, J. Bermudez, J. Weber, and A. Cleve, "Establishing referential integrity in legacy information systems: Reality bites!" in *Int'l Conf. on Software Maintenance (ICSM)*. IEEE Comp. Soc., 2014.

[24] C. Nagy, L. Meurice, and A. Cleve, "Where was this SQL query executed? A static concept location approach," in *Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 580–584.

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

[26] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Int'l Conf. Mining Software Repositories (MSR)*. IEEE, 2013, pp. 207–216.

[27] L. Meurice and A. Cleve, "DAHLIA: A visual analyzer of database schema evolution," in *CSMR-WCRE Software Evolution Week*, 2014, pp. 464–468.

[28] R. Wettel and M. Lanza, "CodeCity: 3D visualization of large-scale software," in *Int'l Conf. Software Engineering (ICSE)*, 2008, pp. 921–922.

[29] M. Goeminne and T. Mens, "Evidence for the Pareto principle in open source software activity," in *Workshop on Software Quality and Maintainability (SQM)*, ser. CEUR Workshop Proceedings, vol. 701. CEUR-WS.org, 2011, pp. 74–82.

[30] J. Ruttan, *The Architecture Of Open Source Applications*, 2008, vol. II, ch. OSCAR.

[31] C. Nagy and A. Cleve, "Mining Stack Overflow for discovering error patterns in SQL queries," in *Int'l Conf. Software Maintenance and Evolution (ICSME)*, 2015.