UNIVERSIDAD NACIONAL DE COLOMBIA

A Framework for Evaluating Maintainability and Performance of Object-Relational-Mapping Tools in Web Application Frameworks.

Danny Alejandro Alvarez Eraso

National University of Colombia

Facultad de Minas, Departamento de ciencias de la computación y de la decisión

Medellín, Colombia

2017

A Framework for Evaluating Maintainability and Performance of Object-Relational-Mapping Tools in Web Application Frameworks

Danny Alejandro Alvarez Eraso

Submitted as partial fulfillment of the requirement to obtain the degree of:

Magister en Ingeniería de Sistemas

Director:

Ph.D. Fernando Arango Isaza

Research Area:

Software Engineering on Web Application Frameworks

Investigation Group:

Grupo de Investigación en Ingeniería de Software

National University of Colombia

Facultad de Minas, Departamento de ciencias de la computación y de la decisión

Medellín, Colombia

2017

A mi mamá, quien se sacrificó durante largos años para hacer este sueño posible. Gracias por toda la dedicación, por aguantar los días de incertidumbre, por los altibajos

… al final lo logramos.

Al profesor Fernando Arango, por las largas horas de trabajo, dedicación y paciencia. Por haber sido más un mentor que un profesor.

A todas las personas que me dieron cariño, amor, fuerza y energía para continuar.

Abstract

Web Application Frameworks (WAF) and Object-Relational Mapping (ORM) tools reduce the time and effort needed to develop high quality applications. However, there are many of them and choosing the best suited for a particular project is not easy. There are comparative studies about these tools, inconveniently, the existing studies do not considerate a domain complex enough that allow us to measure in a more precise way their advantages and disadvantages, also they do not analyze the maintainability related to ORM usage. To contribute in understanding these tools, we propose a WAF-ORM evaluation framework that measures performance and maintainability which we used to compare a group of WAF-ORM technologies. Our study includes a case study that separates queries in three different complexity levels that use both pure SQL and ORM's API. We addressed performance analysis by studying the impact of queries' complexity and database size, leaving all other variables affecting performance constant. Also, as maintainability is harder to measure, we propose a qualitative way to classify WAF-ORM technologies. A comparison based on these aspects is important because understanding performance and architectural benefits of ORMs technologies is helpful for both users and developers of WAF-ORM technologies.

Key Words: object-relational mapping, relational databases, object-oriented languages, class diagram, performance, maintainability, software architecture

Content

Figure List

Table List

Glossary

Abbreviation	Term
WAF	Acronym for Web Application Framework
MVC	Acronym for Model/View/Controller
OOP	Acronym for Object Oriented Programming
ORM	Acronym for Object-Relational Mapping
CD	Class Diagram
IDE	Acronym for Integrated Development Environment
API	Acronym for Application Program Interface
RDBMS	Acronym for Relational Database Management System
OO	Acronym for Object Oriented
JPA	Acronym for Java Persistence API
PC	Personal Computer
ECB	Makes reference to the Entity-Control-Boundary Pattern
POJOs	Plain Java Objects

Introduction

WAFs are widely used today since they reduce time and effort to produce high quality web applications [1]. The notion that WAFs improve developers' productivity is not new. According to [2] frameworks are important development tools as they take advantage of reusing code, design and analysis; being the last two the most rewarding. WAFs offer a layer based architecture and a predefined directory structure to organize files. Among those architectures, Model/View/Controller (MVC) was one of the first widely used, it was developed in the late 70's in Smalltalk-80 as a user interface framework [3]; now, MCV is used in web development (and implemented in many WAFs) as an architectural pattern that focuses on separation of concerns [4].

Most WAFs use Object Oriented Programming (OOP) [5], as it is well suited to represent and give structure to complex data in a way that it is both flexible and maintainable [6]. Moreover, Booch [5] states that:

> "*Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world. This leads to improved maintainability and understandability of systems whose complexity exceeds the intellectual capacity of a single developer or a team of developers*."

However making these data persistent is a problem since OOP traditionally deals with temporary data [6]. A data object lives in RAM and it exists as long as there are pointers to that memory address; when the main program finishes, is stopped or the physical machine is shut down the data disappear.

To address this issue, most WAFs use relational databases as they are good at representing large amount of simple data [6]. There are many options when selecting a relational database, some are free like MySQL [7], MariaDB [8], PostgreSQL [9], and other

products need to be purchased for commercial use like Oracle Database [10] and SQL Server [11]. Every product has different features and implements different SQL standards.

However, the representation of data in databases and OOP languages is different. In accordance with [12], they belong to two different paradigms, each one, with its own abstractions and organizing principles; this is, relational databases are good representing large amounts of simple data and, in contrast with OOP, not so good storing complex objects and their associated relationships [6]. The issues that arise when integrating both paradigms in a single application are referred as *impedance mismatch* [6].

A technique used to integrate those representations is the ORM[13]. They define a map for converting data between those incompatible paradigms, and use an interface to provide an Application Public Interface (API) aimed to perform the basic Create, Read, Update and Delete (CRUD) operations in the OO program.

ORMs are abstraction barriers as defined by Abelson [14]. In fact, using ORMs opens the possibility for developing business classes and databases more independently, as representation mismatches can be solved within the barrier. Consequently the use of an ORM can be an important maintainability factor.

Many WAFs offer a built-in ORM, or offer support for the large amount of off-the-shelf ORMs available online. For example, Hibernate[15] and MyBatis[16] are off-the-shelf ORMs for Java based WAFs, and Doctrine[17] for PHP based WAFs. In contrast, Ruby On Rails[18] and Laravel[19] come out of the box with built-in ORMs, Active Record and Eloquent respectively.

Contrasting off-the-shelf ORMs, there are custom-made ORMs. As off-the-shelf ORMs are not widely accepted, some developers engaged in web development prefer to build their own ORM. This task includes reading and writing the database while solving the impedance mismatch problem.

These two type of ORMs have advantages and disadvantages over each other. Off-the-shelf ORMs reduce the amount of code to develop software as they automatically generate the SQL queries and solve the problem of transforming queries' results into objects.

However, non-optimal off-the-shelf implementations of those two tasks can lead to serious overhead as those processes are not trivial and include complex relational database queries. In contrast, when a developer makes his/her own ORM, he/she is free to apply any efficiency criteria to improve the query performance. Meaning that, automatically generated queries are potentially slower. Consequently, an important ORM quality factor is its capability to use efficiency criteria in order to optimize the queries' performance

Another quality feature is related to the different levels of abstraction achieved by off-the-shelf ORMs. In fact, off-the-shelf ORMs could be bound to the OOP business classes' structure and the database structure. For example, business classes could be forced to represent the same structure as database tables, including foreign and primary key columns. Moreover, the API also affects the level of abstraction by maintaining the object's point of view rather than using table names and foreign keys when crafting SQL queries.

However, high abstraction levels could also threaten the performance, because higher the abstraction, harder to transform the attributes and links between entities in the database to/from attributes and links between objects. Therefore, the queries automatically generated become more complex and sensitive to the optimization mechanisms provided by the ORM. As a consequence, there could be a tension between maintainability and performance between off-the-shelf and custom-made ORMs, being of particular interest, those with a high abstraction level and low features for optimizing the queries.

Under this line of thinking, our research group is interested on evaluating WAF-ORM to classify them and gain understanding on the common concepts of such heterogeneous technologies.

There are studies that compare and classify WAFs focusing on different aspects. In [20] WAFs were compared focusing on documentation, convention over configuration, Integrated Development Environment (IDE), Internationalization, user data input validation and testing. In [21] WAFs were compared focusing on user interface, maintainability, data management and migration, testability, popularity, community, and marketability. In [22] WAFs were compared focusing on navigation rules, validation, internationalization, type conversion, inversion of control, support, and post-redirect features. In [23] WAFs were compared focusing on documentation, MVC architecture, routing, form validation, ORM,

and Testing. Additionally, in [24] authors present a framework to detect redundant data optimizing the data retrieval process when accessing databases using Java based ORM technologies.

Inconveniently, those studies focus on aspects different from performance and maintainability. Moreover, they did not considerate a case study complex enough that allows to see the tension between off-the-shelf and custom-made ORMs which we described before.

In this research we propose a WAF-ORM evaluation framework focusing on maintainability and performance and we evaluate a group of WAF-ORM technologies using it. As described below, the efficiency issue has already been studied by other authors. However there is no study where they considerate a domain with complex relationships and different database sizes. In our study we go one step forward and we consider three reading complexity levels to perform reading experiments with different database sizes. On the other hand, the maintainability issue has not been considered, neither its relation with performance. So, our framework provide a qualitative way of classifying ORMs according to their maintainability benefits.

The relevance of this research consist on gaining understanding about the abstraction benefits and computational cost introduced by ORM tools in a complex domain, closer to real applications. This knowledge helps not only in the task of choosing the best suited for a particular problem but also ***when*** to use them. ORMs are an important comparison aspect since they affect both performance and maintainability. Also, ORMs are heterogeneous, incompatible between them and hard to replace (migrate) for another; so, choosing one wisely is important.

This thesis is presented as follows: in chapter 1 we present the background, in chapter 2 we present the state of the art, in chapter 3 we present our research problem, goals, and methodology, in chapter 4 and 5 we present our WAF-ORM evaluation framework, in chapter 6 we present our results, in chapter 7 the reader can find the details about the implementation that resulted from applying our framework to the WAF-ORM technologies. Finally in chapter 8, we present the conclusions and future work.

1 Background

In this chapter we introduce the technologies framing our research. Below, we present the concepts that support the research in order to define the vocabulary needed to comprehend the rest of the document.

1.1 Object Oriented Programming - OOP

OOP is a programming paradigm widely used nowadays. OOP allows developers to control the natural software systems' complexity by representing the domain elements in a way that is flexible and easy to understand. It supports the reuse [25] of code components whose reliability has already been established [26]. It also improves maintainability [27] by easing the process of modifying or extending the functionality of software components through better data and process encapsulation [25]. All this means that OOP improves overall software quality [28], [29].

Moreover, the object oriented approach is commonly used during all phases of the software development. In fact OOP has shown to reduce the total software life-cycle costs of the development process. Quoting Booch [5]: "*Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world. This leads to improved maintainability and understandability of systems whose complexity exceeds the intellectual capacity of a single developer or a team of developers.*"

Arango [30] states that *"the whole development process must be grounded on the concept that objects serve as a thread that guides the development process, without changing that concept when going from one phase to the next"*. An element of particular importance for our job is stated by Arango [30] who point out that the business classes are the basic elements of software modularization, and, any other form of modularization must respect the boundaries of such classes.

Arango [30] states that object oriented design is grounded on the concepts of classes, encapsulation, inheritance and polymorphism:

- **Classes** are the mean of specifying the common structure and behavior properties of a set of business objects. Class attributes represent the objects' common structure, and the class methods represent the objects' common behavior. The existing links between the objects are represented by object-valued attributes or by inheritance. Classes are the application's abstract data types so that all objects are instances of a class definition.
- **Encapsulation** refers to the mean for standardizing and minimizing the interaction between objects. To do so, encapsulation delimits two sections within the object, the public and the private part. The private part of an object is hidden from all other objects so that the public part is the only way to communicate with it. This allows programmers to take advantage of the objects' structure and behavior without knowing or accessing their private part.
- **Inheritance** is the mechanism to define a class in terms of previously defined classes. This improves code reutilization by allowing developers to define multiple classes that share the properties of a common antecessor. In other words, inheritance makes possible to gradually modify previously defined classes and extend them with new attributes and methods.
- **Polymorphism** refers to the means for defining abstract code trough the object's possibility of defining its behavior without implementation, meaning that descendant classes can have different implementations of the same behavior. However, such behavior must stick to its own rules, even when it is playing the role of one of its parent classes.

1.2 Making data persistent and relational databases

Inconveniently OOP, comes with a problem: program objects live in RAM which is limited and temporal. In other words, OOP programs deal with a reduced number of temporal objects only, which are destroyed when the machine shuts down[6]. OOP per se, offers no mechanism for making data persistent [31]. So, it needs to be completed with a way to access to a database or any mean of persistency so that data exist after the execution of programs.

According to [30], the relational paradigm is the most popular database technology used nowadays. There are already many software products based on the relational model [32]. Relational Database Management Systems (RDBMS) are available both free and under commercial licenses. Among those free, we find MySQL [7], PostgreSQL [9], and more recently, MariaDB [8], which appears as an alternative to MySQL after it was acquired by Oracle. Among the products that need to be purchased for commercial use, we find Oracle Database [10] and SQL Server [11]. Every product has different features and implements different SQL standards meaning that every SQL "dialect" has light variations in syntax and not so light variations in semantics.

The relational paradigm is an application of the set theory and its relationship concept. Under this approach, business objects' attributes are represented by tuples, which are collected forming tables (or relationships in the mathematical terms), where the columns correspond to the attributes and the rows to the objects; as for the business objects links, they are represented by including key attributes in the tuples as objects primary identifications, and as foreign key attributes pointing to the primary keys of the linked objects. Additionally, normalization is a set of tuples organization concepts used to preserve the data, eliminate data redundancy and avoid the introduction of inconsistencies when the data are regularly updated.

To communicate OO programs and relational databases, a language is needed. The most commonly used language, the SQL, is defined using relational operators and, as part of the OOP language, it serves as the main bridge between object oriented programs and the relational databases. In order to use the SQL relational operators, the OO program has to be aware of the tuples data content, and the tuples keys usage. In other words the OO program has to know the way the objects attributes and links are represented in the data base.

1.3 Impedance mismatch

Databases and OOP belong to two different paradigms, each one, with its own abstractions and organizing principles [12]. "*Relational databases efficiently store and manipulate large amounts of simple data*" [6]. However, "*…relational databases are not good at storing*

complex objects and their associated relationships"[6]. The issues that arise when integrating both paradigms in a single application are referred as impedance mismatch [6].

Among the differences in those two paradigms regarding data representation, is worth mentioning those related to data-types and to the representations of objects links. Databases use sequences of primitive elements, as digits and characters, to represent the different data object types, namely integers, floats strings, and timestamps. Most of those sequences have default sizes, but those sizes can be can be changed so that data stored under the same datatype can occupy different amount of bytes in different tables. In contrast, object oriented languages offer a fix set of data types having predefined sizes and binary representations. They offer several types for numbers representing different range of values with different precision, and use built in classes to deal with string of characters occupying arbitrary amounts of RAM. On the other hand, databases represent object links by using primary and foreign keys attributes, as described before, whilst OO programs represent those links as object valued attributes.

1.4 Object Relational Mapping - ORM

To bridge this gap, we need a technique to integrate those two abstractions. Object-Relational Mapping[13] is the name commonly used to refer to the process of mapping objects to relational database tables and vice versa. In [33] ORM tools are defined as middleware configured with a mapping specification which, "*This way, both parts of an application system are largely decoupled from each other.*" Inconveniently, that configuration can't be automatically generated for all kind of applications.

Data stored in databases is queried by the OO program, the result is not returned as OOP objects. Actually they are returned as tables from which OOP objects have to be extracted. For example, PHP can turn a record from MySQL into an associative array using the `fetch_assoc()` method. This is, every row in the returned table must be used to instantiate an object of the corresponding OOP class. This task becomes problematic when complex queries return rows including data belonging to multiple objects. So, in order to simplify the task queries have to be crafted carefully, as well as the modules in charge of automatically turn those complex rows in the corresponding object hierarchy representation. The reader

must note that this task is far from trivial, and it can lead to serious bottlenecks if not implemented carefully.

We classified ORM tools, according to the scope of their applicability, in two types: "*off the shelf*" ORMs and "*custom made*" ORMs. Off-the-shelf ORMs refer to tools developed and conceived as general purpose ORMs capable of map any data from any database to objects of any OOP. On the other hand, custom-made ORMs refer to tools developed and conceived as part of a particular application project and only capable of mapping the database data to the OOP objects of that project.

Off-the-shelf ORMs operates by using a map function relating attributes to table's columns and objet valued attributes to foreign keys. Also, they implement a non SQL interface aimed to perform the basic data manipulation operations (such as Create, Read, Update and Delete - CRUD) without losing the object's point of view.

Custom-made ORMs operate by using explicitly coded SQL queries oriented to perform the needed CRUD operations on the DB, and by using explicitly coded functions oriented to translate the data involved in the SQL queries to/from the data contained in the OOP objects.

1.5 Advantages and disadvantages of ORM Tools

Both off-the-shelf and custom-made ORMs have advantages and disadvantages regarding performance, maintainability, abstraction over the database and developers' productivity and effort.

1.5.1 Advantages

Custom-made ORMs have the opportunity of using optimized SQL queries well casted to the particularities of the specific database in use. This type of ORM can even take advantage from stored procedures that are already compiled inside of the RDBMS, instead of sending the whole SQL string to be compiled. So potentially custom-made ORMs should be faster than off-the-shelf ORMs.

On the other hand, off-the-shelf ORMs increase developer's productivity as they simplify the task of coding to access the DB. They, encapsulate the complexity of querying the database in standardized methods that are easier to use than SQL. This reduces the developing time, reduces the possibility of errors, and avoids recoding when database changes occur. Also, off-the-shelf ORMs aid in the task of migrating from a RDBMS to another as they encapsulate the specific SQL syntax. In the same way, they allow a program to easily connect to different RDBMS from the same application. So potentially of the-shelf-ORMs substantially increase productivity and maintainability.

Additionally, off-the-shelf ORMs function as an abstraction barrier [14] between the relational database and the OO application. This abstraction barrier makes easier to code the application as the OO paradigm becomes the core language to define every component. It prevents or mitigates changes in the class model being propagated to the database and vice versa. Consequently the use of an ORM is an important maintainability factor.

1.5.2 Disadvantages

Using custom-made ORMs can limit the developers' flexibility to switch from one RDBMS to another. The SQL queries it uses are written for a particular RDBMS, meaning that switching it implies recoding the SQL queries. This also means that custom-made ORMs require more time to build and result in products with longer time to market.

On the other hand, off-the-shelf ORMs introduces overhead. They craft the SQL queries to be executed, making it hard or impossible to apply optimization criteria or to work with stored procedures. This threatens the overall application performance.

2 State of the art

Time limit us to test all available ORM and WAF technologies. The numerous technologies available require time to set up for proper testing and the learning curve challenges researchers at all time. Also, there are many ways of designing experiments and test scenarios for gathering performance data; most ORMs and WAFs offer different setups for achieving the same object-relational map, take for example Hibernate that offers XML and annotation based mappings.

As it is difficult to make a clear assessment of the benefits of using the available ORM tools from the point of view of Web Application Frameworks, we decided to focus our research in acquiring a better understanding of the tension between efficiency and maintainability as related to the use of custom-made versus off-the-shelf ORM tools,

To make a valuable contribution in our research, we needed to avoid repeating other authors' work. This implies that we needed a correct assessment and comparison of previous analysis relating to the consequences of using both types of ORMs.

To deliver an unbiased study, Kitchenham et al [34] suggest using a systematic literature review, in order to prevent that researchers' expectations affect the results of the study. We based our review in the guidelines presented in [34], [35] for systematic literature reviews in large groups of researchers. But, we adapted those guidelines so that the review protocol could be defined and executed by a single researcher.

For this review, we defined a protocol based upon the background information already covered in section 1. The protocol included the research questions that the review intends to answer, the search strategy, the inclusion/exclusion criteria, the data extraction strategy, and a synthesis reporting the review of the collected papers, as recommended by the guidelines. Those elements are presented in detail in this chapter.

2.1 Research Questions

The main question driving this systematic literature review is as follows:

- What is the impact of using the WAFs' provided ORM tool in the performance of web applications in different complexity scenarios when accessing relational databases?

This question rise other questions:

- Using the WAFs' provided ORM tools is a good practice? Or on the contrary, using them is harmful for the overall software life-cycle.
- What is the best WAF ORM tool in terms of performance and maintainability?
- The overhead introduced by ORM tools is constant or does it vary with the database size?
- Are the available WAFs' ORM tools effective abstraction barriers between databases and business classes in object oriented applications?
- Do the available WAFs' ORM tools manage to achieve that the object oriented application becomes oblivious of the database structure?

2.2 Defining the search strategy

The search process consisted on a manual search of the main computer science and software engineer journals. Our study included some the journals suggested by Ktichenham et al. [35] -in particular those we have access to-. Moreover, to mitigate the problem of excluding research work not published in those major journals, we also included documents reported in google scholar. In Table 2-1 we list the journals and conference proceedings we used in the search.

Table 2-1: Selected journals and conference proceedings.

Proceeding	Acronym
IEEE Transactions on Software Engineering	TSE
IEEE Software	IEEE SW
Communications of the ACM	CACM
ACM Computer Surveys	ACM sur

ACM Transactions on Software Engineering Methodologies	TOSEM
Software Practice and Experience	SPE
Empirical Software Engineering Journal	EMSE
IEE Proceedings Software (now IET Software)	IET SW
Google Scholar	GS

Also, as suggested by [34], the search terms we used were obtained by decomposing our research question in facets, or key words. In the search terms we present in Table 2-2, the reader can see that we included the keywords JAVA, PHP and Ruby to two of them. This was because, after making some searches, we noted that the language helped to find primary studies. Those three programming languages were selected as they are popular as server-side engines from web applications [36].

Table 2-2: Search terms for the state of the art search strategy.

Term	Term and language
Object Relational Mapping AND Performance	• Object Relational Mapping AND Java • Object Relational Mapping AND Performance AND PHP • Object Relational Mapping AND Performance AND Ruby
Object Relational Mapping AND Overhead	• Object Relational Mapping AND Overhead AND Java • Object Relational Mapping AND Overhead AND PHP • Object Relational Mapping AND Overhead AND Ruby
Object Relational Mapping AND Computational AND Cost	-
Object Relational Mapping AND Performance AND Relationships	-
Object Relational Mapping AND Maintainability	-
Object Relational Mapping AND Database AND Size	-

2.3 Defining the inclusion/exclusion criteria

Peer-reviewed articles published until April 2016 on the following topics, were included:

- ORM tools performance in PHP, Java or Ruby. For example, studies that measure and analyze performance of applications when mapping objects to relational databases using an ORM framework such as Hibernate (Java) or Active Record (PHP, Ruby).
- Case Studies that present details about the implemented classes and database structure, as well as performance comparison from different ORM technologies. For example, response times in stress tests presenting class diagrams or relational diagrams, from a particular software application.
- ORM overhead in PHP, Java or Ruby. For example, studies that compare the overhead introduced by ORM tools when compared to manually coded SQL queries in particular case study.

Articles that can be classified in one of the groups presented below, were excluded:

- Informal studies that do not present implementation details about the classes nor tables involved in the database manipulation tests.
- Duplicate versions of the same study published in different proceedings. As the authors in [35] suggested, when studies have duplicates, we kept the most complete version.

2.4 Data extraction strategy

The data extracted from each study were.

- The source (journal or conference), author name, and date of the publication.
- The ORMs analyzed.
- If manually coded SQL queries were included in the study to compare or analyze performance of ORM queries.
- The case study implementation details, in particular, the class diagram and relational database structure.
- Whether or not the study included complex relationships when performing reading/writing transactions.

2.5 Review Results

2.5.1 About primary studies

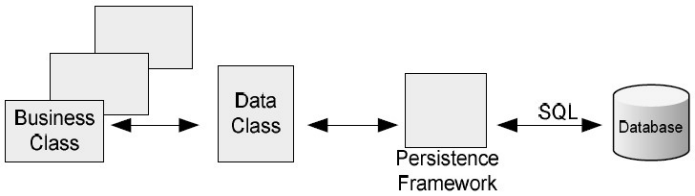
Our review resulted in nine high quality studies that address the performance of ORM tools in web environments. Two of them correspond to a bachelor and a master thesis, the rest were found in the major journals. These studies cover since 2009 to 2016, and highlight the problem that represent overhead in applications that use ORM tools and they offer the initial information for our research.

In [24], the authors state *"ORM mappings introduce redundant data problems (e.g., the needed data in the code does not match with the requested data by the ORM frame- work), which may cause serious performance problems."* So, they propose a framework to automatically detect and fix such redundancies in three case studies, one of them provided by BlackBerry. They concluded that 87% of all transactions contain redundant data, and by detecting and fixing them, the response time is reduced up to 92% and 17% on average.

In [37], the authors evaluate the performance and maintainability of three ORM tools using the goal-driven defined metrics according to the *Goal-Question-Metric* approach which consist on taking the objectives, break them into questions to finally refine them in metrics capable of giving answers [38]. The analyzed ORMs were; NHibernate, a .NET persistence implementation; the Vanatec OpenAccess framework, a commercial mapping framework; and ado.net, a .NET database API. The case of study consists of a read-only oracle database, containing more than 100 tables and used on a daily basis. The results show that ado is the faster one, followed by OpenAccess, and in last place NHibernate. However, the authors do not present the exact class diagram used nor the portion of the table structure that was analyzed. This makes difficult to reproduce and compare the results.

The authors in [39] highlight the role of the "Persistent Framework" (the ORM) as a layer between a database and the business classes (see Figure 2-1). They point out the importance of choosing properly such framework to improve performance. They compare the performance of the OJB and Hibernate persistence frameworks, by read/writing 1000 objects in both a centralized and a distributed DB, concluding that OJB is slower.

Figure 2-1: Persistence framework - from [39].



In [40], the authors present a performance analysis between Hibernate and Eclipselink involving stress testing and heap size measurement. The results give advantage to Hibernate for CRUD operations over 20.000 records; however, they did not analyze the performance for different database sizes or optimized SQL queries.

Similarly, in [41] the authors present a performance comparison between db4o and Hibernate by testing CRUD queries. The conclusion of this study was that db4o is faster than Hibernate, but, the authors stated concerns about using one session per query and left as future work to study the impact in performance of changing this setting.

In [42], the authors compared the performance of db4o and manually coded SQL queries, over a single table with 29 attributes and no primary key. They considered four database sizes ranging 1.000 to 80.000 tuples. The authors concluded that manually coded queries perform better, by a great margin, with the exception of the insertion queries.

The master thesis presented in [43], shows a stress testing evaluation of two PHP WAFs. They evaluated CodeIgniter and CakePHP by developing a website and executing a plan to simulate stress test ranging from 100 to 1100 threads. The testing plan included login in, visiting the dashboard, adding a category, adding a post to that category, do a search to find posts, visit three of those results, visiting the *about* page, and finally log out from the website. Also, the experiments were conducted in a local and a remote environment. However, only CakePHP was tested using the built-in ORM. The authors obtained mixed results for local and remote servers, CodeIgniter performed better in local whilst in live server CodeIgniter started performing better but as the load increased, CakePHP performed better. It seems to us that there could exist non-controlled variables as in theory ORMs introduce overhead and should be slower.

In [44], the authors compare two PHP-based ORM frameworks, Doctrine and DomAr. Authors report performance of select queries in a case study with four tables, but no details

about the use of a WAF is presented. This study uses the API provided by every ORM studied in a series con CRUD operations including a simple relationship: book and author. The authors conclude that Doctrine uses more RAM and performs slower than DomAr at least in a factor of 2.5, their conclusions also allow us to infer that performance is non-linear when increasing the amount of records retrieved in the same database size.

In [45], the authors tested db4o and MySQL by reading and writing on DBs of different sizes. They concluded that for low database sizes db4o performance was closer to MySQL, whilst Hibernate was significantly slower. However, when database size grow db4o's performance rapidly degrades, while Hibernate's and MySQL's response times grow slower. This suggests that even when Hibernate do in fact introduce overhead, it is much more scalable than db4o, being closer to MySQL.

2.5.2 About our research questions

After reviewing the studies presented above, we found that they give incomplete answers for all of our research questions. We summarized those answers in Table 2-3.

Table 2-3: Information obtained by our review.

c	Answer
Using the WAFs' provided ORM tools is a good practice? Or on the contrary, using them is harmful for the overall software life-cycle	Not clear. Authors state that ORMs introduce an abstraction barrier that helps developers to separate data modeling from persistence issues. This separation makes programs more understandable and modular. However, primary studies highlight the difficult on measuring maintainability, authors from [37] explain it is hard to define metrics to study maintainability because it is influenced by many volatile factors such as the developer's experience.
What is the best WAF ORM tool in terms of performance and maintainability?	Hibernate for Java and Doctrine for PHP are the most popular ORM implementations. However, the primary studies offer limited information about maintainability and performance as their case studies are limited to one or two classes.

The overhead introduced by ORM tools is constant or does it vary with the database size?	Not enough information. Primary studies focus on a single database and present little detail about the database size when performing the test experiments.
Are the available WAFs' ORM effective abstraction barriers between databases and business classes in object oriented applications?	Not answered.
Do the available WAFs' ORM tools manage to achieve that the object oriented application becomes oblivious of the database structure?	Not answered.
Which ORM were studied?	Hibernate for Java, Doctrine for PHP and NHibernate for .NET.
WAFs are considered in the primary studies?	Only a few of the primary studies analyze ORMs in a WAF environment.
Is overhead a serious problem?	Not clearly. We found many ORMs being tested under different conditions, some of them present poor performance while others have satisfactory behaviors. It seems that overhead is bound to the ORM implementation
The case studies were presented properly?	A few times. Therefore, reproducing many of them is hard as little detail is presented or the implementation belongs to private companies.
Is there a case study that can be used as reference for our study?	No. Even when studies share common aspects, most of them focus on a single table.

Of particular interest to our research is that the research presented in [39]–[42], has inconveniently focused on a single class/table and does not explore the complexities related to relationships or optimized SQL queries. Those studies also present little implementation details, making difficult to reproduce the experiments. Moreover, even when [43] uses more than one table, the authors did not present the implementation details regarding database access.

Even when the studies presented in [44], and [45] considered relationships, they are reduced to relationships between two classes/tables (flights-airports and cities-departments), and the database sizes were not large enough. So, even when the analyzed ORMs have a satisfying behavior, the evidence is not sufficient to say that it will remain as scalable as SQL queries.

3 Problem and Methodology

The systematic literature review presented in chapter 2 made clear that not all the proposed questions have a satisfactory answer. So, below we present the open issues, and later we proceed to define our thesis problem, goals and methodology. To do so we first state the importance of answering some of the open questions -the thesis statement-, and state the approach we chose to answer those questions -the thesis hypothesis.

3.1 Open issues

After conjecturing that using off-the-shelf ORMs -the one offered by a WAF-, versus using a custom-made ORM -the one coded for the application-, creates a tension between maintainability and efficiency, we analyzed the literature in order to enhance our understanding of the issue.

We found however that still exist many unanswered questions -the open issues. Among them, those relevant to our thesis are listed below.

- A threat to external validity of the studies we reviewed was the case study used for the experiments. Some studies show that ORM tools have a good performance in controlled environments when involving one table and one class model. However, those results might not apply to more realistic environments where queries involve data from multiple tables. As a consequence experiments in the primary studies leave doubts about the queries' behavior when data from complex objects are involved.
- Primary studies we reviewed focused on a single database size, leaving unanswered if performance will be the same when database grows.
- Most of the primary studies ignore the impact of complex relationships when building a case study. We found that, only the studies presented in [44], [45] considered a relational database with a single relationship. This is inconvenient as in medium

sized applications; queries with multiple join sentences are not rare. Moreover, the class diagram from those studies is reduced to one or two classes, which also is a threat to their validity

3.2 Research Problem

To improve the answers to our research questions, we formulated the following research problem:

“There is no clear way of evaluating maintainability, performance and their tradeoff in web applications based on WAF-ORM tools.”

3.3 Thesis statements

Knowing the performance overhead introduced by of the off-the-shelf ORMs available in a WAF, is an important factor when choosing the most suited WAF and ORM for a development project.

It is particularly important to know, how the overhead is related to the complexity of the queries and database sizes.

The knowledge of the referred overhead is also an important decision factor, when evaluating the tradeoff between gaining performance and loosing maintainability, as related to the use of a custom-made ORM.

Understanding the reasons affecting the off-the-shelf ORMs performance, is a key factor in the process of building and improving the ORMs tools.

3.4 Hypothesis

An evaluation approach that includes a case study with different complexity levels and different database sizes, in many WAFs, can give developers, project managers and ORM developers, more precise information about the overhead introduced by ORM technologies from the market when compared to a manually coded ORM.

3.5 Research goals

3.5.1 General goal

Propose and apply a framework for evaluating maintainability, performance and their tradeoff in web applications based on WAF-ORM tools.

3.5.2 Specific goals

- Find the most popular ORM framework implementations for Web Application Frameworks.
- Find the most popular WAF that support the ORMs identified.
- Define a framework for evaluating the tradeoff between maintainability and performance as:
    - A case study composed by a database structure and a Class diagram.
    - Query scenarios over the study case to guide the experiments.
    - Define the database sizes to consider.
    - Define the architectonic guidelines to maximize the maintainability under any WAF-ORM technology.
    - Define the metrics to measure performance and obtain indicators.
    - Define the qualitative measure to assess maintainability.
- Measure and compare the performance and maintainability of the most popular ORM frameworks to an equivalent solution implemented using pure SQL.
- Characterize the ORM-WAF pairs according to the benefits they provide for building maintainable and scalable web applications.

3.6 Methodology.

The construction of our framework for evaluating maintainability and performance occurred in three cyclic phases. In the first phase, we defined the framework's elements for measuring performance. Those elements consisted on a case study to be implemented, a set of three queries with different complexity levels, and a set of metrics to quantify and compare the ORM's performance. In the second phase, we identified The ORM and WAF technologies to be analyzed. By implementing our case study in those technologies we

acquired the necessary insight to refine our third phase where we proposed a qualitative way of measuring the benefits for maintainability when using ORM technologies.

4 Evaluation framework - Performance

We based our performance assessment on a case study, a set of queries, and performance metrics. Our caste study is meant to be implemented multiple times on the WAF-ORM technologies to be analyzed and compared; this is, one for each off-the-shelf ORM, and one using a custom-made ORM. For each implementation, we defined three object trees to be retrieved from a relational database; we called these retrieval strategies the complexity level of queries. Finally, we defined the metrics to guide the experiments and compare performance.

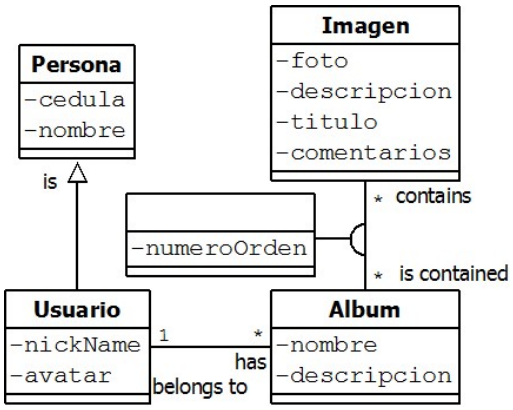
4.1 Case Study

Our case study consisted of an advertisement web application oriented to maintain several photo albums for several clients, being possibly to query the albums and photos for one or more clients.

4.1.1 Class diagram

In Figure 4-1 we present the Class Diagram (CD) of our advertisement web application. We drew the diagram using the notation proposed in [46].

Figure 4-1: Case study class diagram.

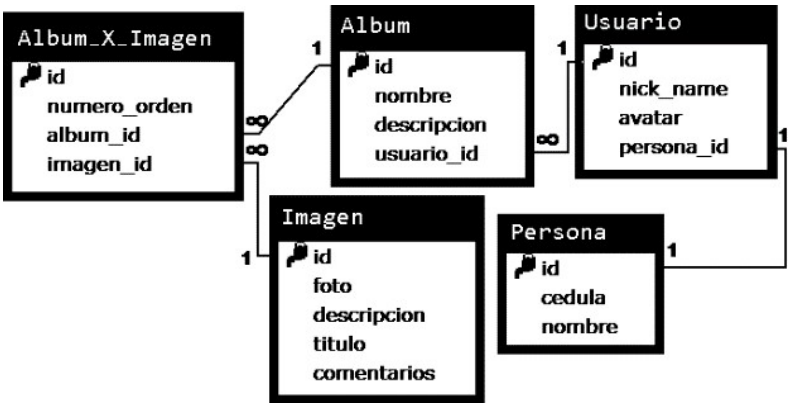


Note that in our application, one client can have several albums, each album can have several photos, and one particular photo can appear in more than one album having a different position in each album.

4.1.2 Database Structure

In concordance to our CD, our database has five tables, one for each class and one intersection table to represent the *many-to-many* relationship. In this way, the relationship attribute *numeroOrden* representing the position of one photo in one album was represented with an extra column in the intersection table. In Figure 4-2 we show our database Entity-Relationship diagram.

Figure 4-2: Entity-Relationship diagram for our case study.

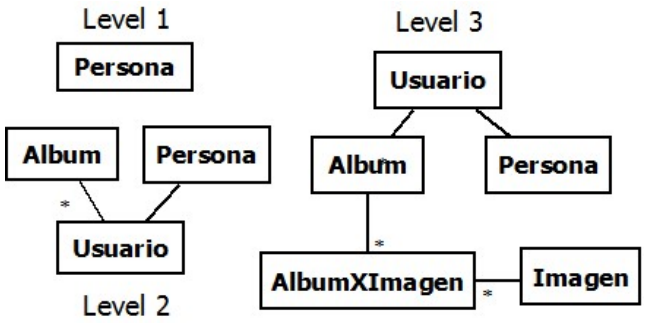


4.2 Complexity level of queries

In our study, we related the query complexity level, to the complexity of the result obtained by reading the database and turning the retrieved data into sets of objects with multiple links. Based on this line of thinking, we defined three query complexity levels, each one associated to one of the diagrams shown in Figure 4-3. Each diagram represents the structure of the set of objects retrieved by the corresponding complexity level query.

Also, all of database queries, with SQL or off-the-shelf ORMs, were filtered by the person's name according to a random alphabetical structure (as in SQL: *"like %"* + *char* + *"%"*). The resulting database readings were to the first 50 root objects with no limit to related ones.

Figure 4-3: Goal object tree for each query level.



4.2.1 Level 1 – Persona

Level 1 is our lowest query complexity level. At this level, we retrieve a list of *Persona* objects. We consider this query complexity level the lowest, because it only reads one DB table, and it only creates a simple object list. This occurs, for example, when we want to retrieve some of the people that are registered in the website.

4.2.2 Level 2 – Usuario and Album

Level 2 is our intermediate query complexity level. At this level, we retrieve a set of *Usuario, Persona* and *Album* objects. This query complexity level is intermediate, because it obtains data from three database connected tables, and has to turn the data into the set of objects connected as shown in Figure 4-3. This occurs when we want to know the albums associated to a user with a given name.

4.2.3 Level 3 – Album and Imagen

Level 3 is our highest query complexity level. At this level, we retrieve a set of *Usuario, Persona, Album* and *Imagen* objects, as well as the image order number (*numeroOrden*, see Figure 4-1). This query level occurs, for example, when we want to know what images are in the albums of some user.

This is the most complex scenario for two reasons. In the first place, increasing the search depth in a database implies accessing more tables, and this is a process that can take a long time if the database contains a large amount of records, or the queries are not

optimized. In the second place, the objectual representation of *many-to-many* relationship between *Album* and *Imagen* needs to include an attribute as part of the relationship.

4.3 Performance metrics

Aligned with the purpose of our research, we took measures to achieve that the impact of query-complexity and database size were the only relevant variables when analyzing the performance of queries. In the first place, we followed the official documentation to setup default configurations for all ORM implementations, for Apache and the RDBMS. In the second place, to achieve the best possible coding practices, we consulted the official documentation and online forums of each WAF-ORM. Additionally, we followed the performance recommendations from [47] regarding annotations. This way we maintained constant all other variables in all ORM implementations we considered.

We found that measuring performance needs special care as the response time varies when running any query multiple times. RDBMS improve the execution plan and give the same result in less time. Moreover, performance is also affected by the cache mechanism abroad all the technologies we used. So, reporting one single experiment iteration it is not enough.

To deal with those time variations, we ran the experiments multiple times and we added all queries a random component. We achieved this by adding a random character in the where clause of all queries implemented according to the definitions of section 4.2, this way we forced the RDBMS to search in different disk segments each time. On the other hand, we did twenty runs for each {implementation - query complexity level – database size} combination using both, the manually coded and the off-the-shelf ORMs. So, those experiments were repeated under three database sizes 100,000, 500,000 and 1’000,000 tuples in the *Usuario/Persona* tables. The number of *albums* and *imagen* tuples was randomly generated between [T – 3T], where T represents the number of tuples in the *Usuario* table.

As data obtained with this method cannot be directly compared, we used the mean response time and the standard deviation. The mean response time is useful to cover the time reduction caused by the optimizing behavior present by default across the used

technologies. Also, the nature of our experiments allow us to use the standard deviation to measure the consistency of the response time from every ORM tool. It is worth mentioning that, this strategy is widely used in the studies we analyzed in the state of the art. The mean response time is interesting since it is an estimate of the time in which the application will respond in a real environment. We assumed to find similar behavior when multiple clients use the case study website at the same time.

4.4 Technological Considerations

Our first attempt to build our case study was done using Windows 10, and later we switched to Ubuntu 15.10 a reliable Linux distribution. In Windows we used wampserver 2.2, Spring MVC and Hibernate only. This study was presented and published in [48], where we found that Hibernate introduces a huge overhead for level 3 queries. We also tested two manually coded SQL queries for level two experiments. These results, and the fact that most web servers run on Linux, raised questions about the external validity of our experiments. To improve the quality of our results, we continued our implementation using Ubuntu, a popular Linux distribution.

One important conclusion of running the experiments in Windows was that queries for level 2 and 3 can be optimized. So we applied those optimizations found to the experiments made latter in Linux.

In both Operating Systems, we used standard hardware and software to run the experiments in. We used a single machine, a Samsung Ultrabook, Intel Core I5-3337U CPU @1.80GHz x 4, 4GB DDR3 RAM, running a fresh installation of the corresponding operating system. We continued on Linux because it is an environment widely used in web servers, and because it is flexible enough to run all of the ORM technologies we identified in section 3.6. In Table 4-1 we show the specific software we used in Linux.

Table 4-1: Technologies used in our case study's implementation

Software	Version
Xampp for Linux	5.5.24-0
PHP	5.5.24
Spring Tool Suite	3.7.2 Release
Tomcat	7.0.34
Hibernate	4.2.2.Final

MyBatis	3.2.8
Doctrine	2.5
CodeIgniter	3.x
Laravel	5.3.2
RubyMine	8.0.3
RubyOnRails	4.2.5

To improve the accuracy of the execution time measurement, we encapsulated the querying process using DAO objects that were implemented as equal as possible across all five WAFs. We used the DAO pattern to encapsulate the process of establishing a connection to the database, querying it, and retrieving the corresponding object list. After doing this, we measured the execution time of those DAOs using two timestamps following the official documentation of each programming language. We placed those timestamps inside every DAO before and after the execution of three steps it is programmed to do. The difference between those two timestamps is the time execution we seek.

Finally, to improve the quality of the reported times in our experiments, and to mitigate any sort of noise affecting them, we followed three main strategies. The first strategy consisted on running all experiments with the same machine conditions. This means that everything was tested the same day, on a single dedicated machine, running the same background process load. The second strategy was to avoid tune-ups that could give advantage to any ORM or SQL queries. We achieved this by using the default configuration for every ORM and WAF as stated in their official documentation. The third strategy consisted on finding a way to deal with the multiple cache mechanisms present in every ORM or WAF implementation, in MySQL and in the Operating System itself; when querying the database, the retrieved data will either be actually read from the hard drive or taken from the already loaded data in RAM. We dealt with those cache mechanisms by randomizing the search space of queries as explained in section 4.3.

5 Evaluation framework - Maintainability

Maintainability in software products can be improved or threatened by modularity[49]. According to [50], maintainability is "*the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.*" So, business classes and design patterns must be carefully designed so that they boost modularity.

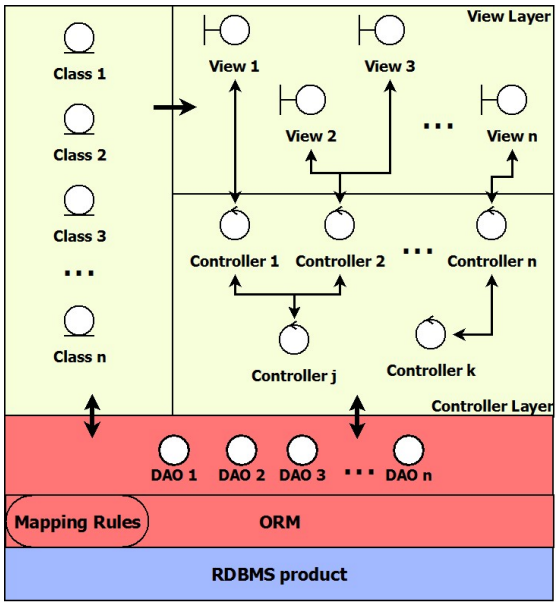
As modularity decisions are important to improve maintainability, the software architecture under any WAF-ORM must achieve high cohesion and low coupled modules [51]. According to [51], cohesion refers to the degree to which elements of a module belong together, those elements that are strongly related, collaborate to achieve a common goal. Consequently, they define a particular functionality. On the other hand, coupling refers to the degree of interdependence between software modules. Cohesion and coupling in software design are correlated, meaning that systems that achieve high cohesion tend to be low coupled as well. Moreover, integrating those two ideas not only improves maintainability but also readability.

However, it is not clear how to achieve high cohesion and low coupling in the analyzed WAF-ORM studies. Even when WAF-ORM technologies benefits from both, OO data type abstractions, and layered code organization principles as the MVC pattern, they integrate those concepts differently. Take for instance the facts that, some WAFs consider business classes and DAOs the same thing, or the fact that business class models contain attributes/getter/setter methods for some WAFs, whilst for others, classes are nearly blank files with nearly no class members.

For this reason, we proposed a software architecture that not only unifies the concepts mentioned above, but also serves as the check list to evaluate the software maintainability achieved by the WAF-ORM technologies we are evaluating. In Figure 5-1, we present a proposal for a unified software architecture for WAF based development. The diagram is

based on, but not limited to, the notation of the Entity-Control-Boundary (ECB) Pattern notation [52].

Figure 5-1 A proposal for a unified software architecture for WAF based development



This architecture have the advantage of making the business classes transversal to the development process, allowing developers to use the OOP as the common language to build all software components. Moreover, in concordance with [30], in this architecture the business classes are the basic element of software modularization, so that the ORM tool, the MCV and DAO patterns, respect the boundaries of such classes.

This is ideal for achieving that objects from the OO application are oblivious of the database structure as the impedance mismatch is solved by the DAO and ORM layers. This makes possible to map any database structure to any class diagram. In our opinion the layered architecture must not superimpose over the OO design and vice versa; they must complement each other.

Our Architecture serves as a target for constructing our photo application. This allow us to run all implementations under the closest conditions possible. To that end, we proceeded as follows:

- We followed the official documentation to setup the WAF and link them to the corresponding ORM.
- We built our photo application following the Model-View-Controller (MVC) design pattern as defined in [3] and the Data Access Object (DAO) pattern.
- We defined a uniform set of controllers with routing mechanisms to execute the application through a web browser.
- We defined a uniform set of DAOs with means for measuring the time necessary to read the database in each experiment.
- We defined uniform set of business classes and a uniform way of representing relationships in the object oriented application. We did this by using string types for primitive attributes from our case study's class diagram and using object valued attributes for relationships.
- We linked the object oriented application with the database tables using the ORM recommended features and executed the experiments.

The reader must note that, implementing the previous architecture in the selected WAFs and ORMs is not trivial, because each WAF offer its own directory structure for code allocation, as well as different pre-elaborated components and helpers.

So, it is possible that the business classes are not correctly represented in the code. Even when WAFs define an architecture, it does not prevent developers from allocating code in an arbitrary way; in other words, WAFs do not necessarily enforce their architecture. This is inconvenient because, when models are not correctly represented in the code, it is easy to break the directive of using the OO design as the guiding thread.

To address those issues we implemented our proposed architecture as following the strategy presented below. Also, as the implementation details may be of interest of certain readers, in section 7 we present the code behind this strategy.

5.1 The business class models - Model layer

We began the development process by coding the business classes in correspondence to the business classes described our class diagram. This way the class diagram provides the common language for the design and the implementation phases of our project. In

concordance to [3], we used this language when referring to the data in the control an view layers, avoiding any reference in those layers to the particular technology of each WAF-in particular the one related to persistence the persistence.

Even if it was not needed in all WAFs, in an effort to make our business classes uniform across all implementations, we coded as class attributes, the primitive attributes of the class diagram, with their correspondent getter and setter methods. We also included, getters encoding the procedures to calculate the derived attributes of the class diagram, and methods to verify the classes constraints (or classes invariants) already identified in the analysis phase. We avoided the use of inheritance.

We represented relationships between two classes with object valued attributes of the related class type in both classes [53]. The relationship's cardinality determines if the attribute is a single object or a collection of objects. Under this optic we can classify the relationships in our case study as one-to-one, one-to-many, many-to-many, and inheritance relationships.

For one-to-one relationships, we included an attribute of the other side class type in both classes. For example, to represent the relationship between Usuario and Persona, we added an *Usuario* typed attribute in *Persona* and a *Persona* typed attribute in *Usuario*.

For one-to-many relationships, we did as in the one-to-one for the one-side of the relationship, whilst for the many-side we used a list-of-objects attribute of the opposite side class type. For example, the one-to-many relationship between Usuario and Album was represented with one *Usuario* typed attribute in Album and one Album-typed list attribute in *Usuario*.

The many-to-many relationship between Album and Imagen classes is more complex as it has an attribute that will be lost if modeled with two typed-list attributes. Given that the many-to-many relationship was solved in the database with an intersection table containing an extra column, we decided to use a new class (*AlbumXImagen*) containing an attribute for numeroOrden. So, to maintain consistency from the point of view of Album and Imagen classes, we included an AlbumXImagen-typed attribute in both of them.

We represented the inheritance relationship between Usuario and Persona using object valued attributes instead of using the object oriented language inheritance features. We did so because, no matter whether the inheritance was modeled as an object valued attribute or extending a parent class, the underlying process for reading data is theoretically the same as it needs to read both tables using the foreign key.

5.2 Controllers - Controller Layer

Controllers are meant to coordinate the interaction between views and models or between controllers[3]. We understand controllers as data and request flow managers, this means that they should receive and redirect all data and requests to the appropriate program module. This implies that controllers must know “who” is in charge of serving a request rather than having the logic to do it by itself. Controllers are also in charge of applying validations to data received from the view, so they can return the corresponding messages to views. However they only apply the restriction by calling the appropriate business class method without knowing the restriction itself.

Consequently we use the controllers to answer specific http requests and receiving data from views, redirecting the action to other controller -in charge of the specific request- or to a DAO -when the database is involved-. Another strategy consists on a controller calling a static method from a class model to execute a DAO. Both strategies fulfill our architecture. So, we implemented the first one.

For our photo applications there is a controller for each ORM. Each one of those contains three methods, one for each complexity level (see section 4.2). Each method reroutes the execution to the DAO in charge of the experiment. Later, the controller returns the experiment results to the browser through a view; in Spring for example controller’s methods vary in one single line, the one responsible for querying the database. Finally, those Experiment results were modeled using a class called *ExperimentMeasuring,* which stores the time interval we are interested in.

5.3 Data Access Objects - Model layer

DAOs encapsulate the storage mechanisms in an application. "*The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components*"[54]. Meaning that, DAOs can be used to separate the model layer from the particularities of the database interface in the WAF. In this way the model layer remains oblivious of the database paradigm or the ORM type.

Under this approach, DAOs constitute an abstraction barrier[14] that benefits the software maintainability. In our applications, DAOs isolate the object-oriented data perception of the controller, model and view layers, from the relational data representation of the RDBMS. This feature allows developers to standardize the interface for communicating with the DB, which in one hand reduces the complexity of the model layer, and on the other hand centralizes the specific database related code in one single point in the WAF's directories structure

We uniformly defined all DAOs containing three methods, one for each complexity level. Each photo application have one DAO, so we implemented six DAOs in total, one for pure SQL, Hibernate and Mybatis; two for Doctrine and Eloquent; and one DAO for Ruby's Active Record. These DAOs measured the time an object, the "*ExperimentMeasuring*" object, coded with the time measuring functions of the WAF language: *Date().getTime()* for Java, *microtime(true)* for PHP and *Time.now* for Ruby. All DAO functions where implemented as follows:

- Instantiating the *ExperimentMeasuring* object to set the experiment's starting time
- Connecting to the database though the WAF API.
- Building the experiment query using the ORM interface.
- Executing the query.
- Retrieving the corresponding object tree
- Use *ExperimentMeasuring* object to set the experiment's ending time
- report the results

5.4 Views – View Layer

The views are in charge of presenting data to the user screen. Views must present the user with the data he/she requested. Those data displayed are contained within objects from the business classes, referred in the views by using the objects public interface; getters and setters.

Our view layer uses simple templates. Each photo application has a single view capable of printing both, the experiment time in milliseconds and the object structure obtained by the experiment from section 4.2. We did not use CSS or JavaScript, every experiment is triggered using a link that redirects to the appropriate controller. The controller returns the experiment results and shows them using the same view, only with the new data. However, the reader must note that time spent on refreshing the browser does not affect the experiment result because the time interval is measured within the DAO method.

5.5 Maintainability metrics

Defining metrics for measuring maintainability is not easy. As [37] explained, maintainability is influenced by many volatile factors such as the developer's experience. So, we assumed that by implementing our case study using the architectonic elements exposed in this chapter, we improved maintainability across all implementations of our photo application. Meaning that, proposing a quantitative metrics for measuring maintainability is out of the scope of this research.

Consequently, we proposed a qualitative metric based whether or not the ORM achieves that the OO application becomes oblivious of the database structure. This metric evaluate three levels of abstraction: Object Oriented ORMs, Table aware ORMs and Object Assembler ORMs. This classification was made because after analyzing ORM technologies we concluded that none of them achieve that the object oriented application is fully oblivious of the database structure.

- ***Object Oriented ORMs****,* are the stronger abstraction barriers we found. They consist on ORMs where developers start from the object's point of view and latter link them with the corresponding table. This type of ORM automatically craft queries,

including queries that imply join clauses by using the name of the related attribute or the relation name without the need of explicitly calling the foreign key names. Those API make that queries are resistant to table column renaming and as a consequence they are more scalable and easier to maintain. Given that Classes define their own attributes and methods, they benefit from data and process encapsulation. However classes and tables have one to one correspondence. This means that even when developers start defining business classes, they must define them taking into account the ORM limitations. This makes developers not totally free to model business classes. In this group we classified Hibernate and Doctrine.

- ***Table aware ORMs*** are intermediate abstraction barriers. They consist on ORMs where developers start defining the database structure and later define empty classes or use the WAF's features to automatically generate them, reducing the abstraction achieved by business classes. Also, the API they provide to the developer requires foreign key names for join based queries. Those queries are harder to maintain because table column renaming implies updating all queries that make reference to that column. As a consequence, queries defined using those API are less scalable than queries defined by Object Oriented ORMs. Moreover, DAOs built with table aware ORMs are weaker abstraction barriers as, even when using them helps to isolate the persistence complexity from the business classes' point of view, require more work and effort to maintain. In this group we classified Eloquent and Rails' Active Record.

- ***Object assemblers ORMs*** are the weaker abstraction barriers we found. It applies to ORMs where the developer must specify the actual query being delivered to the database as there is no API that eases the task of define them. As a consequence these kinds of ORM are more a data mapper or object assembler than object relational mappers. This leaves to the developer all work of defining all the SQL queries making little difference to develop without ORM at all. In this group we classified Mybatis and the custom-made ORM we implemented.

6 Results

As a way of validating our evaluation framework, we selected five ORM-WAF tools and we evaluated them with it. The obtained results show the impact of query complexity in the performance of ORM and pure SQL queries. In each of those five implementations, we executed experiments according to the complexity levels defined in section 4.2. Tests were repeated with different database sizes: 100,000; 500,000 and 1'000,000 tuples. Additionally, every test was executed multiple times, measuring the execution time in every iteration.

We performed each test multiple times because execution time changes. Performance time shows a downgrade trend, when repeating the same experiment multiple times. Those changes occur as a consequence of the cache mechanisms present in WAFs, ORMs and MySQL's service. Even the operating system affects the performance; as when querying the database, the data retrieved will either be actually read from the hard drive or taken from the already loaded data in RAM. Moreover, we deployed our study case in windows over spring [48] and Linux according to section 4.4

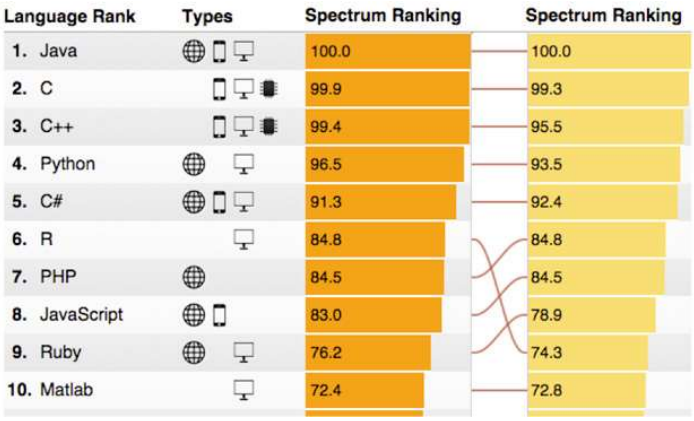
Consequently, we present our results in two steps both for Windows and Linux. The first step presents the experiment raw data using time vs iteration graphs. The second step compares performance relying on the average response time and standard deviation. The average response time is interesting since it is an estimate of the time in which the application will respond in a real environment. We can expect to find similar behavior when multiple clients use the website at the same time. Below we present the strategy used to narrow the WAF-ORM technologies, the data results, performance data analysis and maintainability results.

6.1 What Languages and ORMs are to be included in our analysis?

We narrowed the programming languages to JAVA, PHP and Ruby based on their advantages for our purposes. We needed programming languages that support the level of data and architectural abstractions we proposed in our WAF-ORM evaluation framework. Also, we needed programming languages that support web environments. Moreover, as we want a representative ORM-WAF sample, we included both mature and younger programming languages. Consequently, JAVA was selected as it is a mature language that not only supports OOP but also offers the Java Persistence API (JPA). Ruby was selected as it is a younger language that also supports OOP. Finally, PHP was included because our research aims to contribute in the development of PISIS FRAMEWORK, a PHP based WAF prototype developed by our research group in the National University of Colombia.

This selection allow us to assess the overhead problem across different technologies and determine whether it occurs in all of them, or if it changes with the language, or if its behavior is different in every ORM technology. Our selection was also based on the top web languages among those ranked by the IEEE Spectrum [55], a system that claims be driven by weighting and combining 12 metrics from 10 data sources. In Figure 6-1 we present the spectrum available in June 11, 2016. This information is useful as it suggest which ORMs to analyze.

Figure 6-1: The top ten programming languages, from [55].



Given the time available for this work, we selected five ORM implementations according to its maturity and documentation. This approach is convenient because both, a mature tool

and up-to-date documentation help us to deliver a photo application with the target architecture defined by our WAF-ORM evaluation framework.

Under this approach, we selected Hibernate and MyBatis from Java, Doctrine and Eloquent for PHP and Active Record for Ruby. Also, to implement the manually coded SQL queries, we selected the JPA. This selection is also supported by the Google Trends service. In Figure 6-2 we present the interest over time for Java based ORMs and in Figure 6-3 the interest over time for PHP based ORMs.

Figure 6-2: Google Trends for Java based OMRs until May 2016

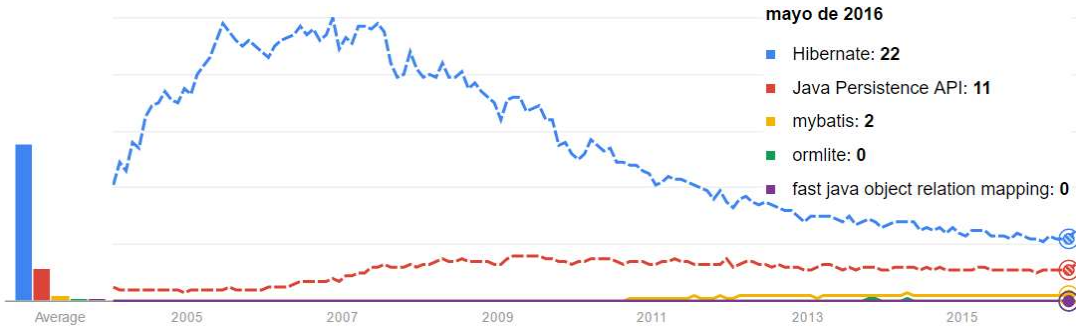


Figure 6-3: Google Trends for PHP based OMRs until May 2016

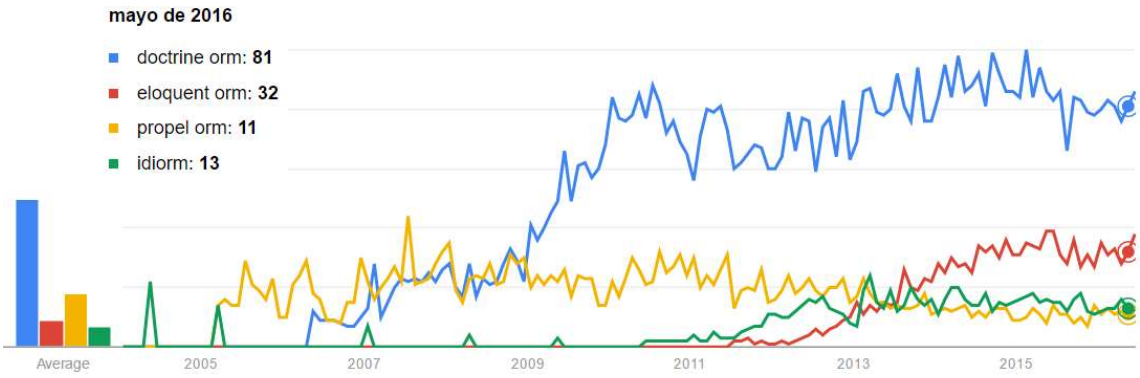
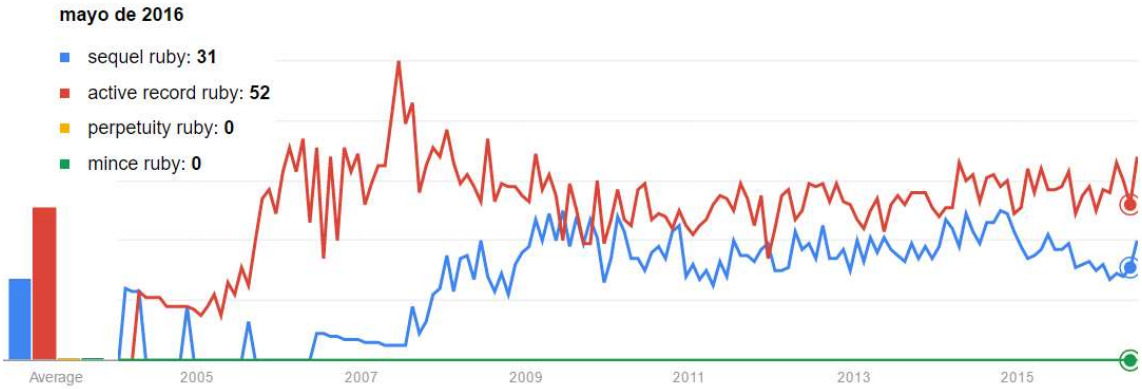


Figure 6-4: Google Trends for Ruby based OMRs until May 2016



6.2 What Web Application Frameworks are to be included in our analysis?

Once defined the ORM implementations to study, we identified a suitable WAF to use it with. So, we selected Spring Framework for Hibernate, MyBatis and JPA, CodeIgniter for Doctrine, Eloquent comes as a part of Laravel, and Active Record comes as part of RubyOnRails. We selected them because either, such ORM is part of it, or because of its maturity and documentation. Moreover, we want to have ORMs both mature and younger to see how ORMs overhead behaves in different technologies. This selection is also supported by the Google Trends service. In Figure 6-5 we present the interest over time for Java based WAFs and in Figure 6-6 the interest over time or PHP based WAFs.

We implemented manually coded SQL queries using a single WAF. As the process of reading the database is not performed by the WAF itself, but by the RDBMS, we tested manually coded SQL queries using Java's JPA in Spring.

Figure 6-5: Google Trends for Java based Web Application Frameworks

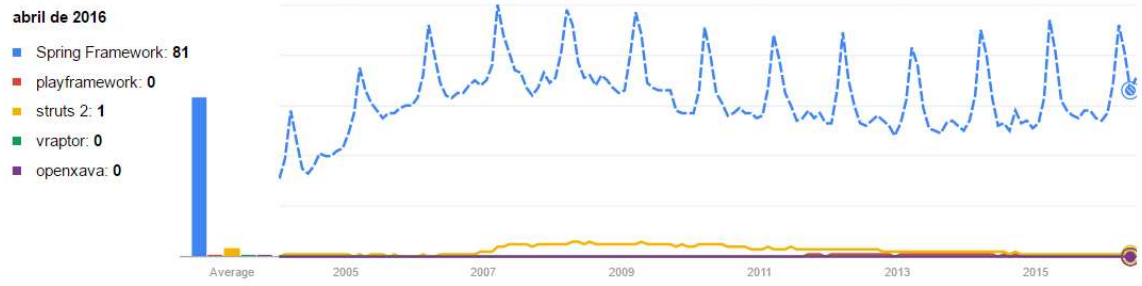
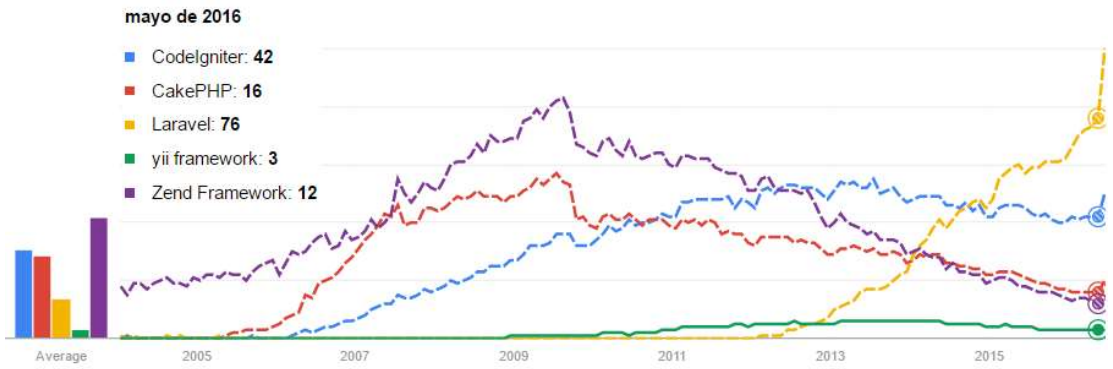


Figure 6-6: Google Trends for PHP based Web Application Frameworks



6.3 Results per complexity level - Windows

Level one results for 100,000 tuples are shown in Figure 6-7, for 500,000 in Figure 6-8 and for 1,000,000 in Figure 6-9.

Figure 6-7: Level 1 queries - 100,000 tuples – Windows

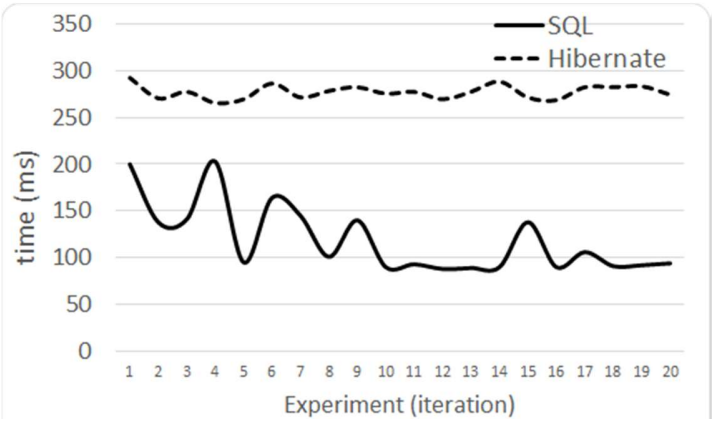


Figure 6-8: Level 1 queries - 500,000 tuples - Windows

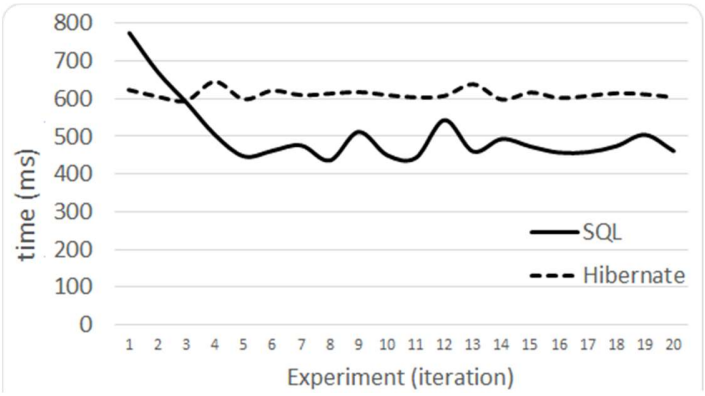
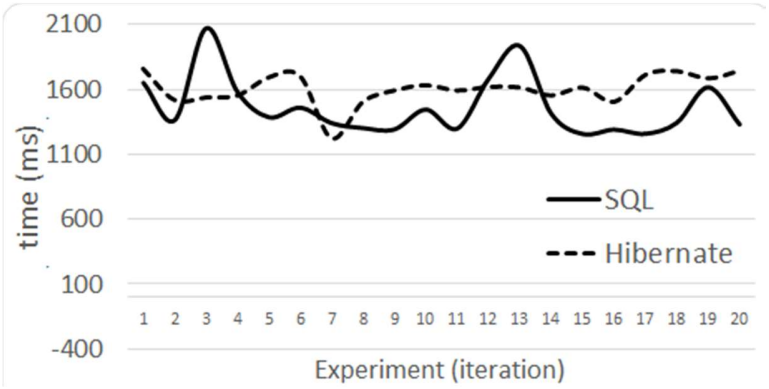


Figure 6-9: Level 1 queries - 1,000,000 tuples - Windows



Level 2 results for 100,000 tuples are shown in Figure 6-10, for 500,000 in Figure 6-11, and for 1,000,000 in Figure 6-12.

Figure 6-10: Level 2 queries - 100,000 tuples - Windows

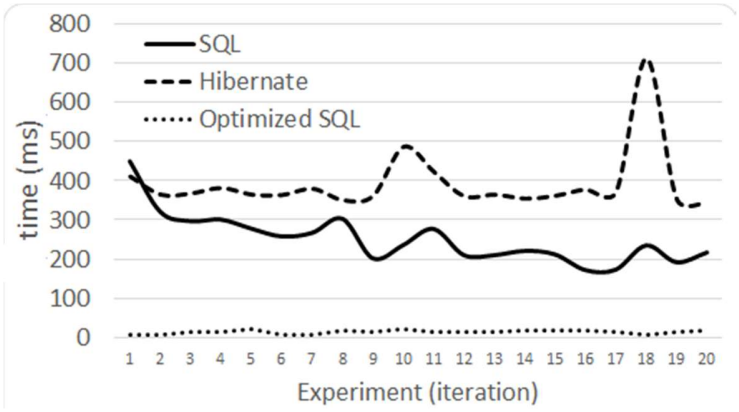


Figure 6-11: Level 2 queries - 500,000 tuples - Windows

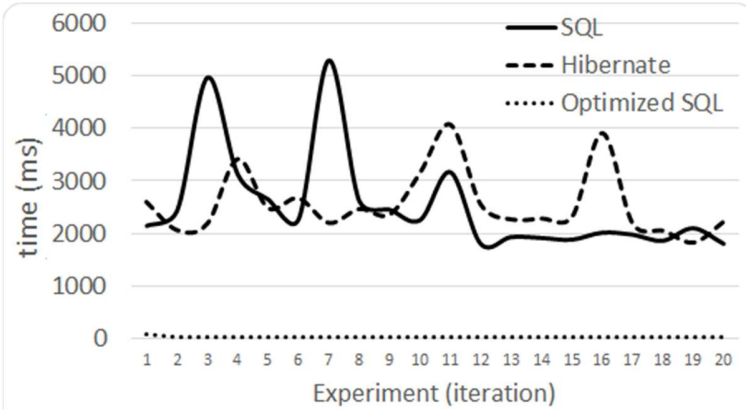
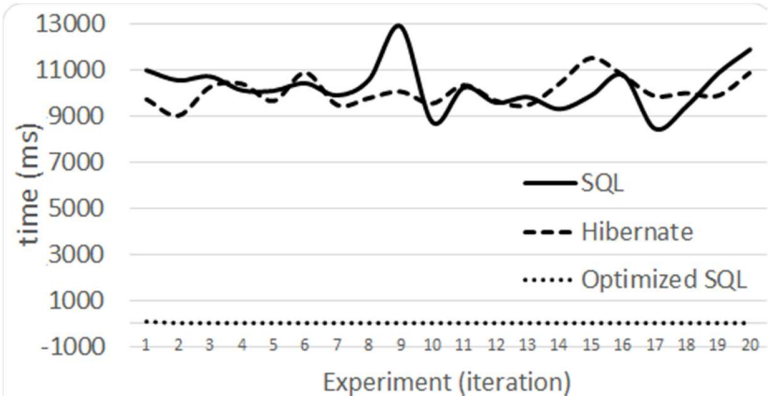


Figure 6-12: Level 2 queries - 1,000,000 tuples - Windows



Level 3 results for 100,000 tuples are shown in Figure 6-13, for 500,000 in Figure 6-14 and for 1,000,000 tuples in Figure 6-15.

Figure 6-13: Level 3 queries - 100,000 tuples - Windows

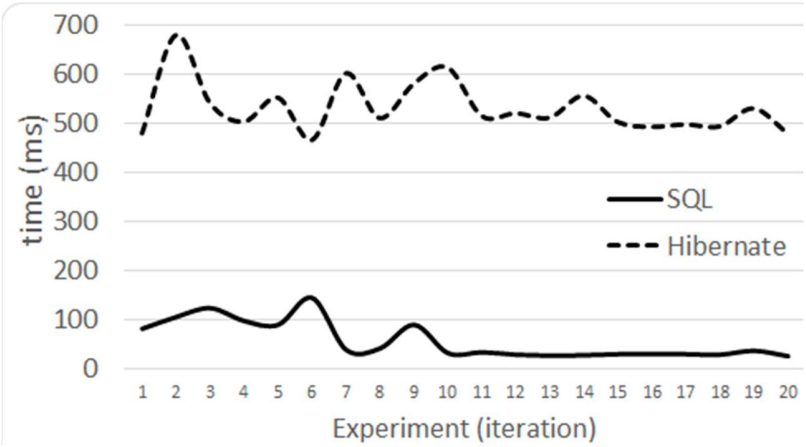


Figure 6-14: Level 3 queries - 500,000 tuples - Windows

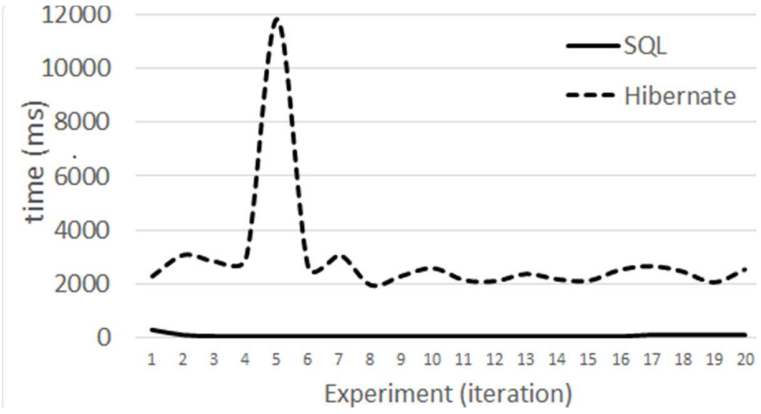
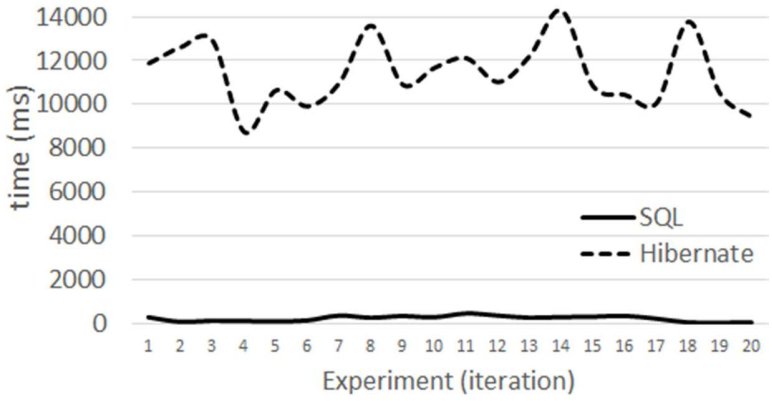


Figure 6-15: Level 3 queries - 1,000,000 tuples - Windows



The mean response times for the three reading levels are presented in Table 6-1. Rows marked with "2*", in the column level/tuples correspond to MySQL optimized level 2 response times.

Table 6-1: ORM mean response times.

Level /tuples	Hibernate (ms)	SQL(ms)
1 / 100,000	277	119
1 / 500,000	611	504
1 / 1,000,000	1,605	1,465
2 / 100,000	392	252
2*/ 100,000		13
2 / 500,000	2,574	2,530
2* / 50,000		18
2 / 1,000,000	10,084	10,274
2* / 1,000,000		17
3 / 100,000	531	56
3 / 500,000	2,910	56
3 / 1,000,000	11,419	218

6.4 Results per complexity level in Linux

Level one results for 100,000 tuples are shown in Figure 6-16, for 500,000 in Figure 6-17 and for 1,000,000 in Figure 6-18.

Figure 6-16: Level 1 queries - 100,000 tuples

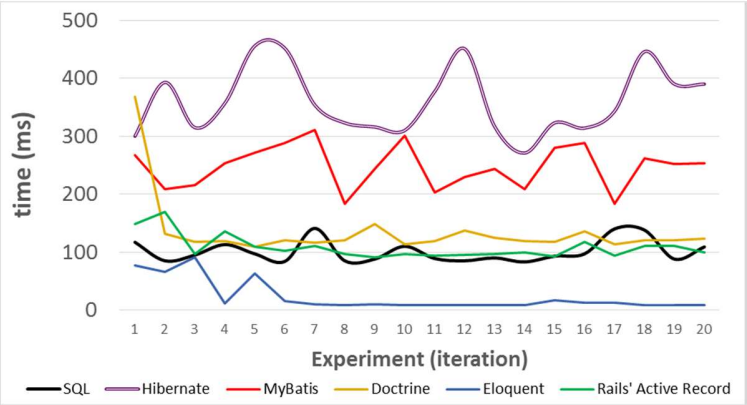


Figure 6-17: Level 1 queries - 500,000 tuples

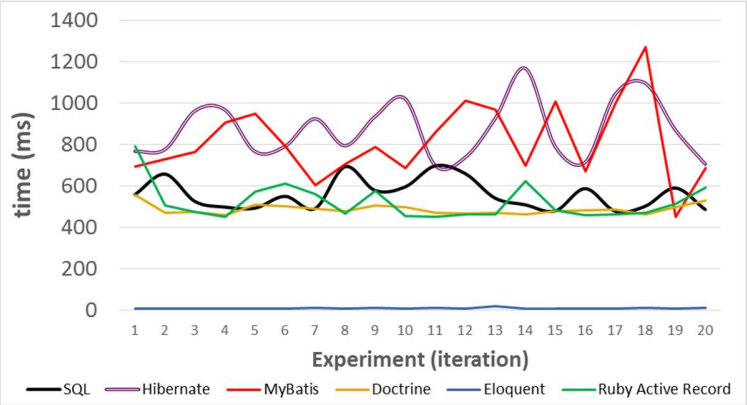
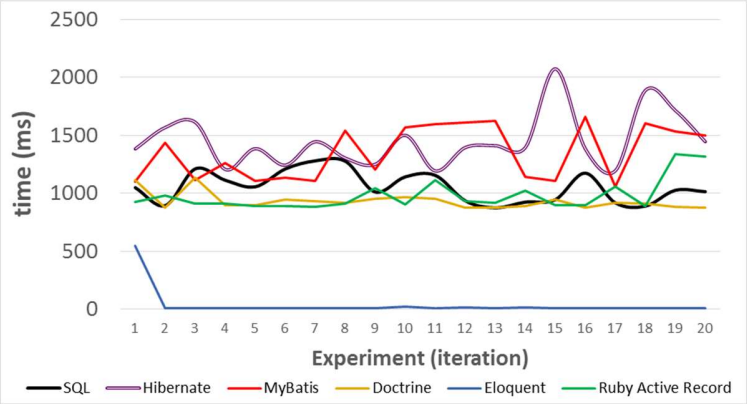


Figure 6-18: Level 1 queries - 1,000,000 tuples



Level 2 results for 100,000 tuples are shown in Figure 6-19, for 500,000 in Figure 6-20, and for 1,000,000 in Figure 6-21. Similarly, level 3 results for 100,000 tuples are shown in Figure 6-22, for 500,000 in Figure 6-23 and for 1,000,000 tuples in Figure 6-24.

Figure 6-19: Level 2 queries - 100,000 tuples

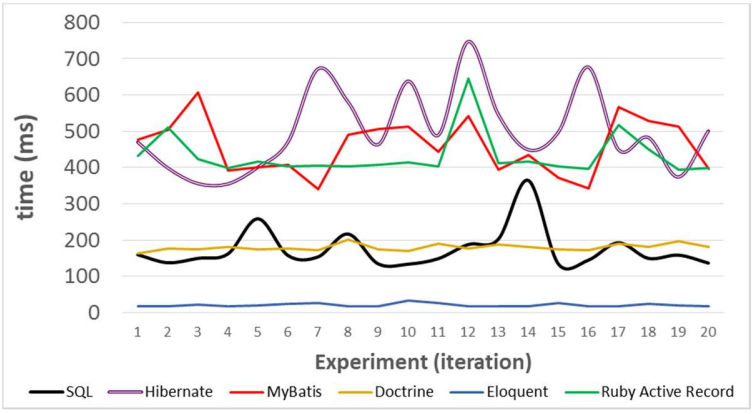


Figure 6-20: Level 2 queries - 500,000 tuples

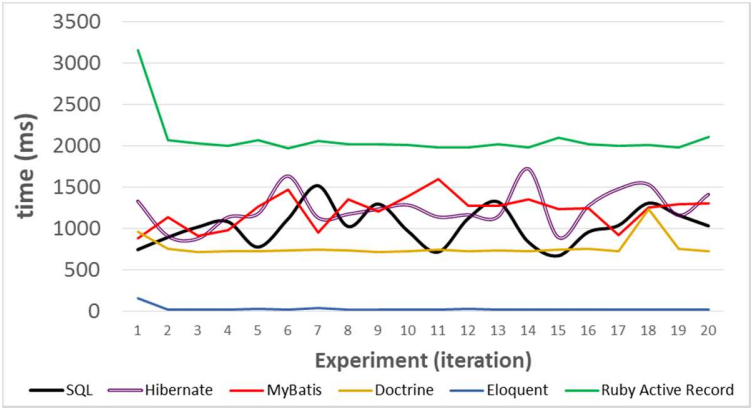


Figure 6-21: Level 2 queries - 1,000,000 tuples

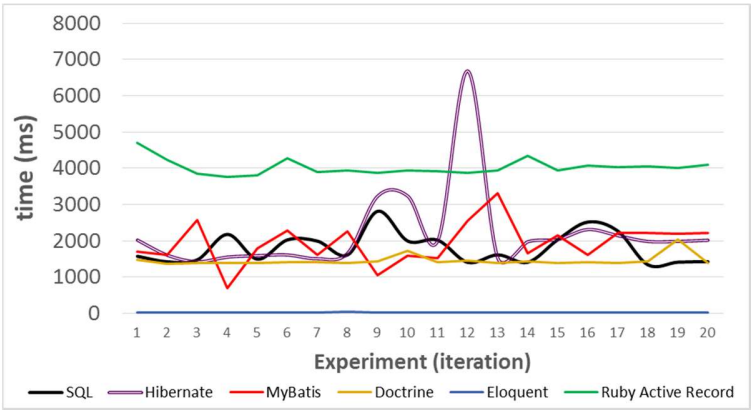


Figure 6-22: Level 3 queries - 100,000 tuples

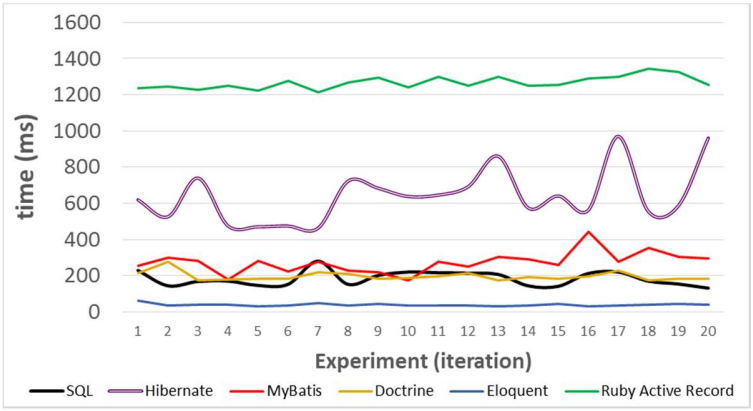


Figure 6-23: Level 3 queries - 500,000 tuples

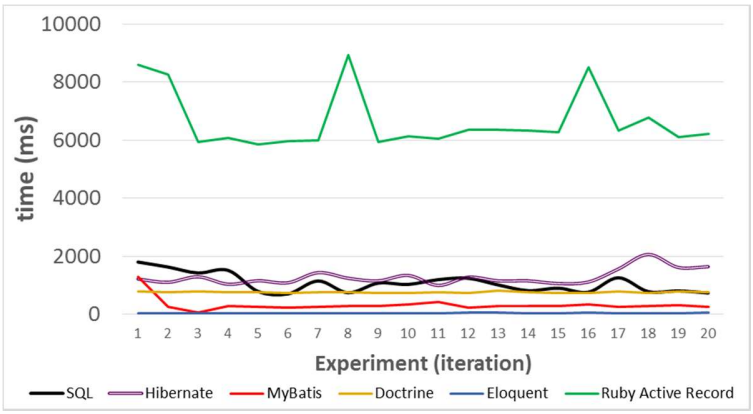
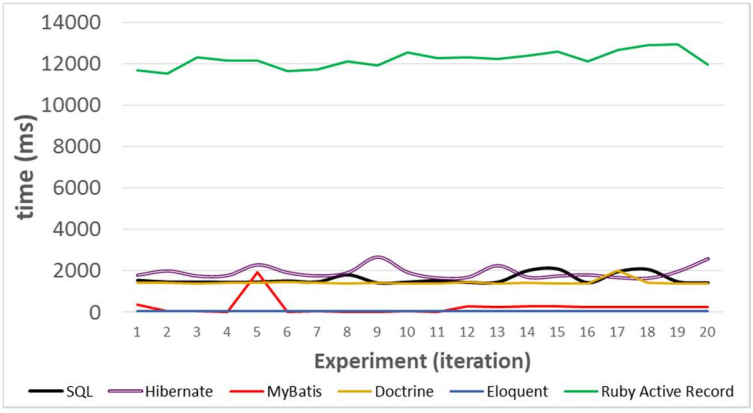


Figure 6-24: Level 3 queries - 1,000,000 tuples



The mean response times for the three reading levels are presented in Table 6-2. Results for Eloquent are partial because we were not able to define the query trough the API.

Inconveniently, it is not clear from the documentation how to add constrains to the where clause referring joined columns, same constrains that are needed to obtain the required object tree for level 2 and level 3 queries.

Table 6-2: ORM mean and standard deviation for response times in Linux

Level/ tuples (thousands)	SQL (ms)		Hibernate (ms)		MyBatis (ms)		Doctrine (ms)		Eloquent (ms)		Ruby AR (ms)	
	M	SD	M	SD	M	SD	M	SD	M	SD	M	SD
1/100	101.35	19.4	359.9	56.9	247.2	37.8	134.9	55.7	23.2	26. 8	108.35	20.7
1/500	559.35	72.0	873.5	139.0	812.6	184.9	488.4	24.1	10.35	2.3	522.8	86.6
1/1000	1053.9	133.01	1448.55	230.5	1348.55	230.2	930.5	73.1	36.7	119.9	980.4	135.5
2/100	174.5	55.3	500.95	111.5	458.5	76.3	179.95	9.1	-	-	432.65	60.9
2/500	1029.75	222.7	1240.25	228.7	1214.85	195.8	770	121.7	-	-	2078.4	256.8
2/1000	1798.55	423.7	2201.4	1166.5	1936.4	575.9	1451.2	156.7	-	-	4024.3	220.1
3/100	184.55	39.9	643.3	150.4	273.85	58.8	197.75	25.5	-	-	1266.9	34.8
3/500	1064.85	327.3	1281.05	263.1	323.3	239.4	756.3	20.9	-	-	6649.05	1018.3
3/1000	1599.8	240.2	1928.3	300.3	250.4	410.8	1443.2	133.3	-	-	12225.35	400.8

6.5 Data Analysis

In this analysis we present the knowledge we obtained by applying our WAF-ORM evaluation framework to five implementations of our photo application. Our framework allowed us to assess the tradeoff between maintainability and performance introduced by ORM tools across multiple programming languages and WAF-ORM tools both mature and younger.

Using this strategy we found different behavior across the technologies used which allowed us to classify WAF-ORM tools according to performance and maintainability. We present these findings together with conjectures about the possible causes of unexpected behavior detected. However, we clarify that the validation of such conjectures requires further research about the implementation details for each WAF-ORM technology and in consequence they are out of the scope of this research. So, these conjectures are based on three principles:

- Whether or not, the ORM crafts more than one query to read the required data from the relational database.
- Whether or not, the ORM optimizes the query before delivering it to the RDBMS product.
- Whether or not, the ORM implements additional cache mechanisms to those offered by the operating system.

As our experiments were made in windows and later in Linux, below we present the analysis for each case.

6.5.1 Windows

For Windows, the mean response times presented in Figure 6-25 and Figure 6-26 corresponding to level 1 and level 2 are similar. This suggests that in those cases, overhead is not a big issue. However, that performance gap grows faster in level 3 being more severe when Hibernate queries the relational database corresponding to 1'00,000 tuples.

In fact, for levels one and two, the mean response time shows that the performance of Hibernate and non-optimized SQL queries is similar; in the best case MySQL is only 160 ms faster than Hibernate. This behavior was already described by [45]. Additionally, when going from half million records to one million, we can see that Hibernate and non-optimized SQL queries' response times are strongly non-linear; they grow three and almost five times respectively.

On the other hand, authors in [45] suggested that their work is susceptible of optimization. Our results for optimized level two and level three queries show a severe gap between the performance of Hibernate and SQL (see Figure 6-26 and Figure 6-27). In Level two, MySQL is 10,066.35 (58,525%) ms faster that Hibernate for the database corresponding to 1'000,000 tuples. In Level 3, the behavior is similar as MySQL is 11,201 (5,139%) ms faster. There is a big difference in this case.

Figure 6-25: level 1 mean response time as function of the database size - Windows

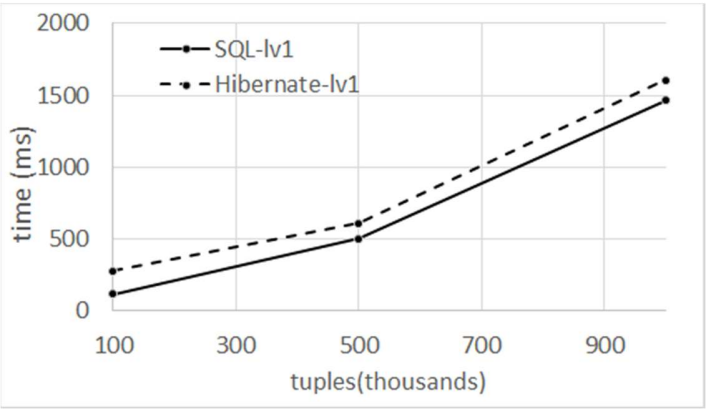
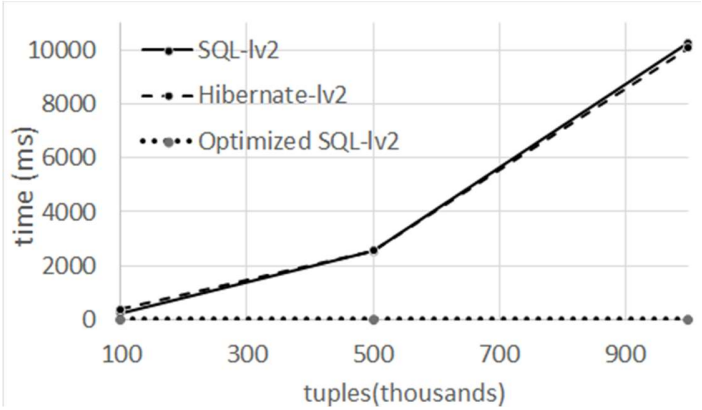
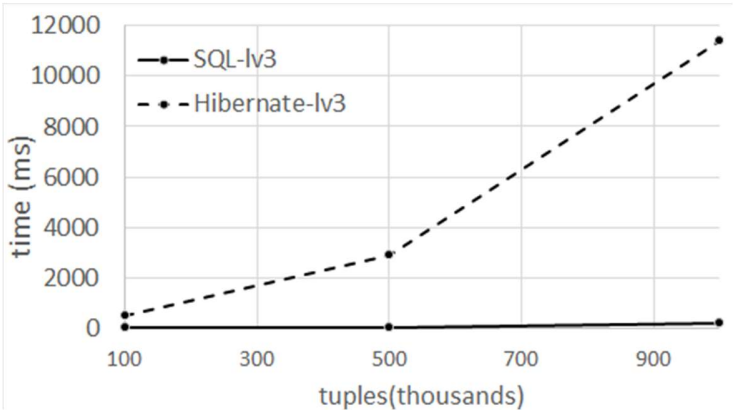


Figure 6-26: level 2 mean response time as function of the database size - Windows



Based on the gap between optimized and non-optimized level 2 queries, we can conjecture that the overhead is caused by the inability of Hibernate for finding the optimum query to retrieve the required object tree in complex transactions. In fact, we could not identify a good way of optimizing level three queries using Hibernate as there is not official HQL documentation regarding this topic. So, to the best of our knowledge, we made sure that Hibernate Level 2 and 3 queries minimized unnecessary database accesses. As for the non-linearity of the performance, more research is needed to explain such behavior.

Figure 6-27: Level 3 mean response time as function of the database size - Windows



Finally, another interesting finding was that hibernate reports doing 26 queries to obtain the results for our level-three-complexity scenario. In Figure 6-28 we show a fragment of the Spring console output of those undesirable queries.

Figure 6-28: Level 3 average time in function of the database size - Windows

```
INFO : co.edu.perf.controller.HomeController - Welcome home! The client lo
INFO : co.edu.perf.controller.ReadScenarioController - Welcome home! The c
Hibernate: select this_.id as id1_4_2_, this_.avatar as avatar2_4_2_, this
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
Hibernate: select usuario0_.id as id1_4_0_, usuario0_.avatar as avatar2_4_
```

6.5.2 Ubuntu – Linux

When we switched to Linux technology, we used MariaDB as an alternative to MySQL producing two interesting changes in performance. The first one consisted on a reduction of the overhead introduced by Hibernate, making the response time grows slower and closing the gap with the SQL's curve. This behavior was also found for Doctrine, Eloquent and Mybatis. The second change is not so satisfactory, under MariaDB, mean response time for manually coded SQL queries was slower; for instance, in MySQL level 3 query, with one million tuples the average response time was 218 milliseconds, whilst the exact same query in Ubuntu was had an average response time of 1599 milliseconds. In Figure 6-29, Figure 6-30 and Figure 6-31 we can see the average response times for level 1, 2, and 3 queries respectively.

Figure 6-29: Level 1 mean response time as function of the database size - Ubuntu

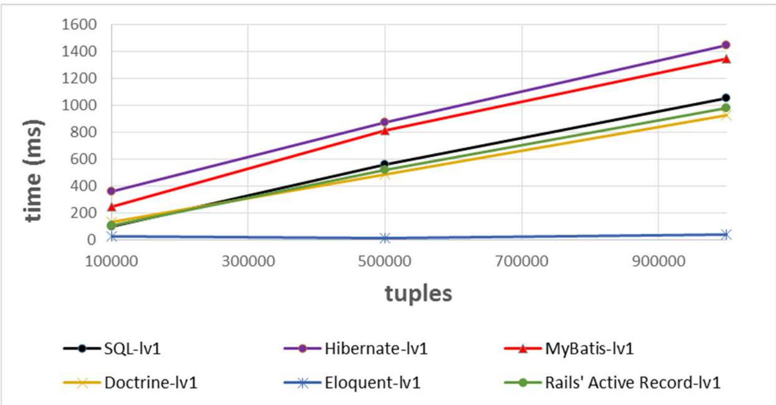


Figure 6-30: Level 2 mean response time as function of the database size - Ubuntu

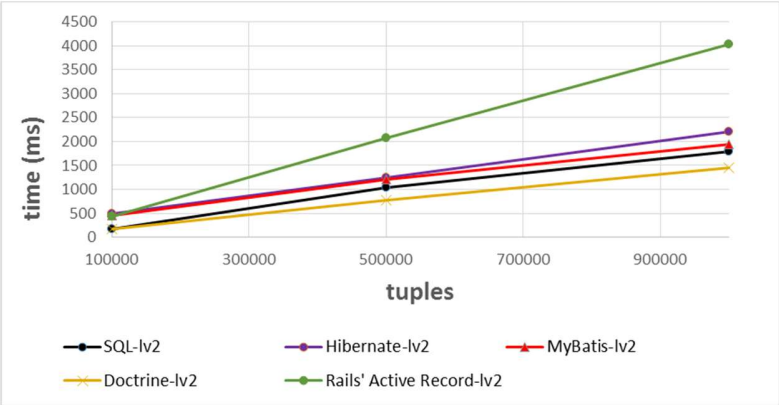
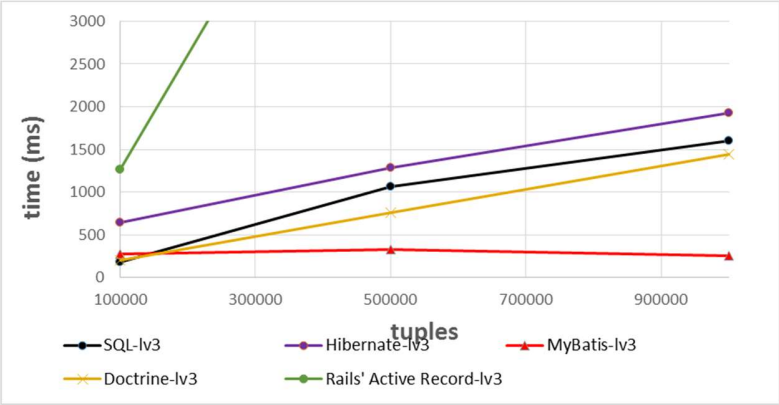


Figure 6-31: Level 3 mean response time as function of the database size - Ubuntu



Our findings suggest that Hibernate and Mybatis; the JAVA based ORMs, can be grouped together lightly above SQL's curve, the PHP based ORMs can also be grouped together below SQL's curve, and in the extreme sides Rails' Active Record is always the slower and Eloquent is the fastest one.

Additionally, as ORMs keep closer to SQL, performance seems less sensible to the database size because the rapid growing response times and the non-linearity we found in windows, now disappeared. This was probably caused by the query optimization mechanism offered by MariaDB.

On the other hand, even when Eloquent had a surprisingly fast behavior for level one queries, the API it provides was limited in our experiments. We were not able to implement our level two and level three queries under the same constrains applied to other ORMs; the documentation is limited an makes no references to the definition of constraints in the tables

that appear in the *join* or *"where"* part of statements. So as far as time allowed us to dig-in, we couldn't find a simple way of solving this. For those reasons, the scope of this work is not enough to fully explain Eloquent's performance and further research is needed to measure performance under Eloquent's features.

Mybatis' performance is also surprising. Even when this tool requires the manually coded SQL query as input, its response time is faster than manually coded SQL queries. Mybatis seems more a data mapper than an object relational mapper because SQL queries must be defined by the developer. This is opposed to what other ORMs we studied do; once defined the class models and the database structure, the ORM automatically crafts the queries.

We found that Rails' Active Record had the slower performance. In level two, the response time nearly doubles Hibernate's the second slowest. Additionally, in level three Rails' Performance is off the chart, reminding to us the same overhead Hibernate introduced in Windows.

Under this scope, we can conjecture that, Hibernate still has difficulties optimizing the query as its performance is similar to that from windows. The gap reduction between Hibernate and SQL can be explained by the fact that SQL is much slower in our Ubuntu implementation. For the PHP based ORMs we can conjecture that, performance benefits from a lower abstraction level which makes the object oriented application bound to the database structure. In other words, PHP based ORMs seem to sacrifice flexibility to gain performance. Finally, the reasons behind Rails and Laravel performance are not obvious. More research is needed to explain the way those two ORMs use cache, craft queries and transform the results into the corresponding object tree.

7 WAF Implementation

7.1 WAF and ORM setup

The setup of each ORM is different, even when target application is the same. Some of the ORMs we selected are part of a WAF and others are plugins that must be attached to the WAF. Below we present the default configuration, implemented following the official documentation.

7.1.1 Spring Framework

Spring framework can be downloaded from the official page and it uses Maven to extend its functionality. Spring plugins, or dependencies, managed by Maven are configured in a file called *pom.xml*. We created two MVC projects to setup the database connection using Hibernate, MyBatis and pure MySQL.

7.1.1.1 Hibernate

We configured this project in two steps. The first step consisted on importing the main required plugins, or dependencies: the MySQL connector, the Hibernate core and the Spring ORM, in Figure 7-1 we show the key part of the code that achieves that. The full configuration file is presented in appendix A. The second step consisted on setting up the dependency injection in the *servlet-context.xml* file so that the controllers and DAOs have a wire to access the database. We did this by creating three main beans[56], one for the MySQL jdbc connection driver, for the session factory support, and the last one to add the query transaction support (see Figure 7-2). The full *servlet-context.xml* file is presented in appendix B.

Figure 7-1: Main dependencies used in Spring for Hibernate and MySQL – pom.xml

```
...<artifactId>mysql-connector-java</artifactId>
       <version>5.1.10</version>
...<artifactId>hibernate-core</artifactId>
       <version>4.2.2.Final</version>
...<artifactId>spring-orm</artifactId>
       <version>3.1.1.RELEASE</version> ...
```

Figure 7-2: Main Beans used in Spring for Hibernate and MySQL - servlet-context.xml

```
...<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
...<bean id="mySessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
...<bean id="transactionManager"

class="org.springframework.orm.hibernate4.HibernateTransactionManager">...
```

7.1.1.2 MySQL

We reused the definitions of Figure 7-1 and Figure 7-2 to setup the experiments accessing the database using jdbc. So, we defined an additional module in the Hibernate Spring Project to use the corresponding bean and access the database through the *myDataSource* bean only.

7.1.1.3 MyBatis

The configuration required for MyBatis is similar to Hibernate's since both implementations were made in Spring framework. The same two steps were made, first defining the dependencies to be included (*pom.xml*) and later defining the controller's access wires (servlet-context.xml), in Figure 7-3 and Figure 7-4 we present the respective key codes that achieve that. Both files are fully presented in appendixes C and D.

Figure 7-3: Main dependencies used in Spring for MyBatis – pom.xml

```
...<artifactId>mysql-connector-java</artifactId>
       <version>5.1.10</version>
...<artifactId>mybatis</artifactId>
       <version>3.2.8</version>
...<artifactId>mybatis-spring</artifactId>
       <version>1.1.1</version> ...
```

Figure 7-4: Main Beans used in Spring for MyBatis - servlet-context.xml

```
... <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
... <bean id="mySessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">  ...
... <bean id="transactionManager"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager">
```

7.1.2 CodeIgniter and Doctrine

CodeIgniter can be downloaded from its official page, however Doctrine is not included with it. To install Doctrine we manually installed composer and used the configuration from Figure 7-5, later we manually copied the Doctrine source files scattered through the composer download directory into the CodeIgniter source files under *\application\libraries\Doctrine\ directory*. Also, CodeIgniter requires a PHP script to load the Doctrine class, this script loads all the ORM components and reads the database access credentials defined in the \application\config\database.php file. We named that script Doctrine.php and we placed it under *\application\libraries\ directory (see the full file in appendix E).*

Figure 7-5: Composer json file for Doctrine

```
{    "require": {
        "doctrine/orm": "2.5.1"
}  }
```

7.1.3 Laravel and Eloquent

Setting up Laravel and eloquent is an easy task, however it can be challenging for developers with no practice using the command line. Out of the box, Laravel includes the Eloquent ORM, however, the composer command line must be used to both download Laravel and set up a project to work with. We created a project using the commands shown in Figure 7-6. Finally, we configured the database access using the MySQL section under the *\config\database.php* file.

Figure 7-6: Laravel Setup using the command line

```
composer global require "laravel/installer"
laravel new performance
```

7.1.4 RubyOnRails and Active Record

Setting up Rails is challenging the first time. Windows users would have a bad time since there is little support; the official documentation is limited about handling errors that arise under this operating system. In Linux, we managed to complete the installation as presented below.

The main problem we addressed concerns the lack of MySQL support in an out of the box Rails installation. Rails' gems were not able to find the MariaDB service running from our XAMPP installation under */opt* directory. Inconveniently, just installing the support using the "`gem install mysql2`" command while the MariaDB service is running, resulted in the error shown in Figure 7-7. Even when the Ubuntu installation was fully updated, it seems that some dependencies need to be installed. We fixed the dependencies issue using the command `sudo apt-get install libmysqlclient-dev`, and manually wired Rails to find our XAMPP installation with the command shown in Figure 7-8.

We were able to build the `mysql2` gemFile after we configured our Rails application to use the service we just installed. We did that in the */applicationRoot/config/database.yml* file by changing the development configuration as shown in Figure 7-9, this allows Rails to use the appropriate MySQL socket.

Figure 7-7: Out of the box Rails error when installing the MySQL support

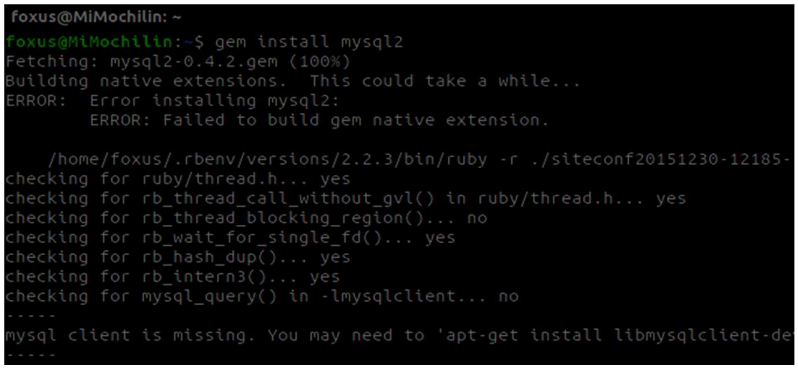


Figure 7-8: Fixing the Rails installation to work with MySQL service

```
sudo env ARCHFLAGS="-arch i386" gem install mysql --with-mysql-
dir=/Applications/XAMPP/xamppfiles/lib/mysql --with-mysql-
lib=/Applications/XAMPP/xamppfiles/lib/mysql/ --with-mysql-
include=/Applications/XAMPP/xamppfiles/include/mysql/
```

Figure 7-9: Database Configuration for Rails – database.yml

```
development:
  adapter: mysql2
  encoding: utf8
  database: performance
  pool: 5
  username: root
  password:
  socket: /opt/lampp/var/mysql/mysql.sock
```

7.2 Controllers and routing

7.2.1 Spring

Controllers for the three Spring implementations are almost identical. They all belong to the Spring MVC template, variating only on the line that routes the http request to a controller's method and the line that calls a DAO method to access the database. In Figure 7-10 we present the general template that defines the controller for Spring implementations, where the *@RequestMapping* annotation links the controller being defined with one unique URL contained in its "*value"* attribute, the *<readLvlScenarioURL>* portion of that URL refers to one complexity level of the query being invoked by the http request, *<readScenarioDAO>* refers to the DAO being used, and *<readLvlScenario()>* refers to one of the three complexity levels defined in see section 4.2.

Figure 7-10: Spring standard template for controller methods

```
@RequestMapping(value = "/ReadScenario/<readLvlScenarioURL>", method =
RequestMethod.GET)
public String readLvlScenario(Locale locale, Model model) {
  logger.info("Welcome home! The client locale is {}.", locale);
  Date date = new Date();
  DateFormat dateFormat =
            DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG, locale);
  String formattedDate = dateFormat.format(date);
  model.addAttribute("serverTime", formattedDate );

  // Call to the DAO method for accessing the database
  ExperimentMeasuring experiment = <readScenarioDAO>.<readLvlScenario()>;
  model.addAttribute("experiment", experiment);
  return "home"; }
```

7.2.2 CodeIgniter

We defined the CodeIgniter controller following the default routing mechanism. CodeIgniter uses the controller name and the method name as part of the browser URL. In Figure 7-11

we present the general template that defines the controller methods for this implementation; again, *<$readScenarioDAO>* and *<readLvlScenario()>* have the same meaning as above.

7.2.3 Laravel

To wire http requests to the desired controller method, we used the \app\Http\routes.php file to define three routes, one for each complexity level (see Figure 7-12). In Figure 7-13, we present the general template that defines the controller methods for this implementation.

Figure 7-11: CodeIgniter standard template for controller methods

```
class ReadScenarioDoctrineController extends CI_Controller {
  public function readLv () {
    $readScenarioDAO = new ReadScenarioDoctrineDAO();
    $readScenarioDAO->setDoctrineEntityManager($this->doctrine->em);

    // Call to the DAO method for accessing the database
    $experiment = <$readScenarioDAO>-><readLvlScenario()>;

    $data['models'] = $experiment->getModels();
    $data['elapsedTime'] = $experiment->elapsedTime();

    $this->load->view('homeReadingsView', $data);}
   ...}
```

Figure 7-12: Laravel routes.php file for our photo application

```
Route::get('readScenario/readLv1Scenario','ReadScenarioElocuentController@readLv1');
Route::get('readScenario/readLv2Scenario','ReadScenarioElocuentController@readLv2');
Route::get('readScenario/readLv3Scenario','ReadScenarioElocuentController@readLv3');
```

Figure 7-13: Laravel standard template for controller methods

```
class ReadScenarioElocuentController extends Controller {
...public function readLv() {
   $readScenarioDAO = new \App\Http\Daos\ReadScenarioEloquentDAO();
   // Call to the DAO method for accessing the database
   $experiment = <$readScenarioDAO>-><readLvlScenario()>;

   echo "Elapsed Time in milliseconds: ";
   var_dump($experiment->elapsedTime());
   return view('homeReadingsViewLv3')->with('models',$experiment->getModels());}
```

7.2.4 RubyOnRails

To wire http requests to the desired controller method, we used the \config\routes.rb file to define three routes, one for each complexity level (see Figure 7-14). In Figure 7-15, we present the general template that defines the controller methods for this implementation.

Figure 7-14: RubyOnRails routes.rb file for our photo application

```
Rails.application.routes.draw do
  get 'read_scenario/readlv1', to:'read_scenario#readlv1'
  get 'read_scenario/readlv2', to:'read_scenario#readlv2'
  get 'read_scenario/readlv3', to:'read_scenario#readlv3'  ...
```

Figure 7-15: RubyOnRails standard template for controller methods

```
class ReadScenarioController < ApplicationController
  def readlv
    @dao = ReadScenarioRailsDAO.new()
    # Call to the DAO method for accessing the database
    @experiment = @dao.readlv()
  end
```

7.3 Class models, relationships and the database tables

Given that our five ORMs do not allow linking classes to multiple tables, we decided to code a class for every database table and one for the relationship between A*lbum* and *Imagen* Consequently, we defined {Album, Imagen, Usuario, Persona and *AlbumXImagen*} classes. Also, to represent the primitive data from the database tables, we used private String attributes. Finally, we included an id attribute as most ORMs need an identification strategy.

7.3.1 Hibernate

Hibernate can establish a link between a Java class and a database table by using annotations. Annotations in the class definitions link them to the corresponding database table while annotating attributes link such attributes with the corresponding table column, or with another object according to relationship defined by the foreign key.

Thus, we linked our CD's class models with our database tables using *@Entity*, *@Table, @Id*, *@GeneratedValue* for Hibernate identification strategy, and *@Column* for the primitive class members. In Figure 7-16, we present Album class as an example.

Figure 7-16: Album class mapping using Hibernate

```
@Entity @Table(name="album") public class Album {
    @Id @GeneratedValue     private Long id;
    @Column(name="nombre")  private String nombre;...   }
```

For one-to-one and one-to-many relationships we used *@OneToOne, @OneToMany and @ManyToOne annotations.* Also, for all cases we defined the inverse relationship using the

@*JoinColumn* annotation. In Figure 7-17 we present point a of view of *Usuario* class toward its relationship with *Persona* and *Album*. Likewise, in Figure 7-18, we present the inverse side of the relationship from the *Album* class' point of view. The reader must note that the owning side of the relationship uses the @*JoinColumn* annotation to tell the ORM the name column that serves as foreign key, the inverse side of the relationship uses the *mappedBy* attribute to tell the ORM the name of the attribute that owns the relationship.

Figure 7-17: Usuario's point of view of the @OneToOne and @OneToMany relationships

```
@Entity @Table(name="usuario")public class Usuario {
    ... @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
        @JoinColumn(name="persona_id")private Persona persona;

        @OneToMany(fetch = FetchType.LAZY, mappedBy = "usuario", cascade =
CascadeType.ALL)
        private List<Album> albumes;...}
```

Figure 7-18: Album point of view of the inverse relationship with Usuario - Hibernate

```
@Entity @Table(name="album") public class Album {
    ... @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name="usuario_id")private Usuario usuario;   ...}
```

We implemented the solution to the many-to-many relationship issue, with @OneToMany and @*ManyToOne* annotations. In Figure 7-19-A and Figure 7-19-B, we present Album's and Imagen point of view of the relationship which was bridged using the AlbumXImagen class' mapping presented in Figure 7-20.

Figure 7-19: Album and Imagen point of view of the many-to-many relationship

A	`public class Album {`
	`...@OneToMany(mappedBy = "album",  cascade = CascadeType.ALL)`
	`private List<AlbumXImagen> albumXimagenList;  ...  }`

B	`public class imagen {`
	`...@OneToMany(fetch = FetchType.LAZY, mappedBy = "imagen",  cascade =`
	`CascadeType.ALL)`
	`private List<AlbumXImagen> albumXimagenList;`

Figure 7-20: AlbumXImagen bridging relationship - Hibernate

```
@Entity @Table(name="album_x_imagen")public class AlbumXImagen {
  @Id @GeneratedValue          private Long id;
  @Column(name="numero_orden") private String numeroOrden;

  @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
  @JoinColumn(name="album_id") private Album album;

  @ManyToOne(cascade = CascadeType.ALL) @JoinColumn(name="imagen_id")
                               private Imagen imagen; ...}
```

7.3.2 MyBatis

MyBatis class definition did not use annotations, instead we used Plain Old Java Objects (POJOs). However, as the attribute distribution is similar to Hibernate's, we omit the java classes.

To define the class mapping, MyBatis needs xml files organized using *<select>* and *<resultMap>* tags. The <select> tags are used to define SQL queries and the data type of the result, if the resulting query contains data from multiple classes a result map must be used to map such data to the corresponding classes.

As this process is manual, we defined a select tag for each complexity level in our case study. For level one, we adapted the SQL query from the pure SQL solution, see Figure 7-21, the reader must note that the result type for this query is automatically converted to Persona Class. For level two, we adapted the SQL query from the pure SQL solution, see Figure 7-22-A, and mapped the result using the resultMap from Figure 7-22-B. Similarly for level three, we adapted the SQL query from the pure SQL solution, see Figure 7-23-A, and mapped the result using the resultMap from Figure 7-23-B.

Figure 7-21: Level 1 select - MyBatis

```
<mapper namespace='co.unal.perfimybatis.entity.mapper.Persona' >
  <select id="selectPersons" parameterType="java.lang.String" resultType="Persona">
    <bind name="pattern" value="'%' + _parameter + '%'" />
      SELECT id AS "id", cedula as "cedula", nombre as "nombre"
      FROM persona WHERE nombre like #{pattern} ORDER BY nombre DESC LIMIT 0, 50
  </select>    ...
```

Figure 7-22: Level 2 query and result mapper - MyBatis

A - Select

```
<select id="lv2UserSelect"
parameterType="java.lang.String"
resultMap="lv2UserMap">
  <bind name="pattern"
  value="'%' + _parameter + '%'" />
    SELECT axi.*,
    al.id AS album_id, al.nombre AS
    album_nombre, al.descripcion
    FROM (SELECT p.id AS person_id,
       p.nombre AS person_nombre,
       p.cedula as person_cedula,
       u.id as id, u.avatar as avatar,
       u.nick_name as nick_name
       FROM usuario u
       LEFT JOIN persona p
            ON p.id = u.persona_id
       WHERE p.nombre like #{pattern}
       ORDER BY p.nombre DESC LIMIT 0, 50
    ) axi
    LEFT JOIN album al
      ON al.usuario_id = axi.id
</select>
```

B - resultMap

```
<resultMap id="lv2UserMap"  type="Usuario">
  <id property = "id" column = "id"
      javaType = "java.lang.Long"
      jdbcType = "INTEGER"/>
  <result property="avatar"
column="avatar"/>
  <result property="nickName"
          column="nick_name"/>
   <association property = "persona"
               javaType = "Persona">
     <id property="id" column="person_id"/>
     <result  property= "nombre"
               Column = "person_nombre"/>
     <result property = "cedula"
               Column = "person_cedula"/>
  </association>
  <collection property ="albumes"
              ofType ="Album">
     <id property="id" column="album_id"/>
     <result  property = "nombre"
               Column  = "album_nombre"/>
     <result  property = "descripcion"
               Column  = "descripcion"/>
  </collection></resultMap>
```

Figure 7-23: Level 3 query and result mapper - MyBatis

A – select

```
SELECT uxp.person_id,
 uxp.person_nombre,
uxp.person_cedula,
 uxp.id, uxp.avatar, uxp.nick_name,
 al.id AS album_id, al.nombre AS
 album_nombre,al.descripcion as
 album_descripcion, axi.numero_orden,
 axi.id as albumximagen_id, i.id as
 imagen_id, i.foto as imagen_foto,
 i.descripcion as imagen_descripcion,
 i.titulo as imagen_titulo,
 i.comentarios as imagen_comentarios
FROM ((  SELECT u.id, u.avatar,
   u.nick_name,
   p.id AS person_id, p.nombre AS
   person_nombre, p.cedula as
   person_cedula
   FROM usuario u
   LEFT JOIN persona p
        ON p.id = u.persona_id
   WHERE p.nombre like #{pattern}
   LIMIT 0, 50 ) uxp)
LEFT JOIN album al
     ON al.usuario_id = uxp.id
LEFT JOIN album_x_imagen axi
     ON al.id = axi.album_id
LEFT JOIN imagen i
     ON axi.imagen_id = i.id
ORDER BY uxp.person_nombre DESC
```

B - resultMap

```
<resultMap id = "lv3UserMap" type = "Usuario">
     ...
  <association property = "persona" .../same
  </association>
  <collection property = "albumes"
              ofType = "Album">
    <id property = "id" Column = "album_id"/>
       ...
    <collection property ="albumXimagenList"
                ofType ="AlbumXImagen">
      <id property="id"
column="albumximagen_id"/>
      <result property = "numeroOrden"
               column = "numero_orden"/>
      <collection property = "imagen"
                 ofType = "Imagen">
        <id property="id" column="imagen_id"/>
        <result property = "foto"
                 column = "imagen_foto"/>
        <result property = "descripcion"
                 column =
"imagen_descripcion"/>
        <result property = "titulo"
                 column = "imagen_titulo"/>
        <result property = "comentarios"
                 column =
"imagen_comentarios"/>
      </collection>
    </collection> </collection>
</resultMap>
```

7.3.3 Doctrine

Our Doctrine implementation follows the same strategy as what we did with Hibernate. PHP class files contain the same attributes for all the columns in the database that are not related to relationships. Moreover, Doctrine uses a similar way of marking PHP classes with the metadata about the database structure by putting annotations in PHP documentation blocks. So, we used *@Column* annotation for primitive data type attributes and *@id* and *@GeneratedValue* for the id attribute. So, we omit the code regarding PHP class and jump to relationships.

For one-to-one and one-to-many relationships we used *@OneToOne, @OneToMany and @ManyToOne annotations.* In Figure 7-24, we present the point of view of *Usuario* class toward its relationship with *Persona* and *Album*. Likewise in Figure 7-25, we present the inverse side of the relationship from the *Album* class's point of view, where the owning side of the relationship is indicated by the *@JoinColumn* annotation. Also, the *inversedBy* attribute represents the attribute that maps the inverse relationship in the linked PHP class.

Figure 7-24: *Usuario*'s point of view of the *@OneToOne* and *@OneToMany* relationships

```
@<?php namespace models;
/** @Entity @Table(name="usuario") */ class Usuario {
   /** @Id @Column(type="integer") @GeneratedValue               */ private $id;
   /** @Column(type="string", length=140, name="nick_name")      */ private $nickName;
   /** @Column(type="string", length=140, name="avatar")         */ private $avatar;
   /** @OneToOne(targetEntity="Persona", inversedBy="usuario")
     * @JoinColumn(name="persona_id", referencedColumnName="id") */ private $persona;
   /** @OneToMany(targetEntity="Album", mappedBy="usuario")      */ private $albumes;
...}
```

Figure 7-25: Album point of view of the inverse relationship with Usuario - Doctrine

```
<?php namespace models;
/** @Entity @Table(name="album") */ class Album {
   /** @Id @Column(type="integer") @GeneratedValue               */ private $id;
   /** @Column(type="string", length=140, name="nombre")         */ private $nombre;
   /** @Column(type="string", length=140, name="descripcion")    */ private
$descripcion;
   /** @ManyToOne(targetEntity="Usuario", inversedBy="albumes")
       @JoinColumn(name="usuario_id", referencedColumnName="id") */ private $usuario;
...}
```

We implemented the solution to the many-to-many relationship issue with @OneToMany and *@ManyToOne* annotations. In Figure 7-26-A and Figure 7-26-B, we present Album's and Imagen point of view of the relationship which was bridged using the AlbumXImagen class' mapping presented in Figure 7-27.

Figure 7-26: Album and Imagen point of view of the many-to-many relationship

A	<?php namespace models; /** @Entity @Table(name="album") */ class Album { .../** @OneToMany(targetEntity="AlbumXImagen", mappedBy="album") */ private $albumXimagenList;... }
B	<?php namespace models; /** @Entity @Table(name="imagen")*/ class Imagen { .../** @OneToMany(targetEntity="AlbumXImagen", mappedBy="imagen") */ private $albumXimagenList; ... }

Figure 7-27: AlbumXImagen bridging relationship - Doctrine

```
<?php namespace models;
/** @Entity @Table(name="album_x_imagen") */ class AlbumXImagen {
  /** @Id @Column(type="integer") @GeneratedValue */ private $id;
  /** @Column(type="string", length=140, name="numero_orden") */ private $snumeroOrden;
  /** @ManyToOne(targetEntity="Album", inversedBy="albumXimagenList")
      @JoinColumn(name="album_id", referencedColumnName="id") */ private $album;
  /** @ManyToOne(targetEntity="Imagen", inversedBy="albumXimagenList")
      @JoinColumn(name="imagen_id", referencedColumnName="id") */ private $imagen;...}
```

7.3.4 Eloquent

By following the documentation and Eloquent's mapping behavior, we see that PHP classes under Laravel and Eloquent have no attributes but only methods. Contrasting what we saw on JAVA, this ORM exploits the flexibility of PHP objects for dynamically adding attributes to automatically load objects following the structure of the database. Also, methods are used to define the relationships with other classes by using the inherited API from Laravel's standard Model.

For one-to-one and one-to-many relationships we used *hasOne(), hasMany() and belongsTo() methods.* In Figure 7-28, we present the point of view of *Usuario* class toward its relationship with *Persona* and *Album*. Likewise in Figure 7-29, we present the inverse side of the relationship from the *Album* class's point of view.

Figure 7-28: Usuario's point of view of the one-to-one and one-to-many relationships

```
class Usuario  extends Model {
   public function persona(){
      return $this->belongsTo('App\Http\Models\Persona', 'persona_id', 'id');}
   public function albums(){
      return $this->hasMany('App\Http\Models\Album');} ...}
```

Figure 7-29: Album point of view of the inverse relationship with Usuario - Eloquent

```
class Album extends Model{
   public function usuario(){
      return $this->belongsTo('App\Http\Models\Usuario', 'usuario_id', 'id');}   ...}
```

We implemented the solution to the many-to-many relationship issue, with *hasOne*(), *hasMany()* and *belongsTo()* methods. In Figure 7-30-A and Figure 7-30-B, we present Album's and Imagen point of view of the relationship which was bridged using the *AlbumXImagen* class' mapping presented in Figure 7-31.

Figure 7-30: Album and Imagen point of view of the many-to-many relationship

A
```
class Album extends Model{
   public function albumXImagenes(){
      return $this->hasMany('App\Http\Models\AlbumXImagen');}   ...}
```

B
```
class Imagen extends Model {
   public function albumXImagen() {
      return $this->hasOne('App\Http\Models\AlbumXImagen', 'id', 'imagen_id');}}
```

Figure 7-31: AlbumXImagen bridging relationship - Eloquent

```
class AlbumXImagen extends Model {
   public function Album(){
      return $this->belongsTo('App\Http\Models\Album', 'album_id', 'id');}
   public function imagen(){
      return $this->belongsTo('App\Http\Models\Album', 'imagen_id', 'id');}   }
```

7.3.5 RubyOnRails' Active Record

Similarly to Eloquent, classes for RubyOnRails' Active Record use blank classes that contain nothing but the relationship definitions with other classes. However, RubyOnRails' Active Record's documentation strongly emphasizes convention over configuration to the point that, it does not even need the foreign key names or table names to work. We used the *self*-pointer to override those conventions, and setup the mapping to work with the same database structure defined for other implementations.

For one-to-one and one-to-many relationships we used *hasOne(), hasMany() and belongsTo() methods.* In Figure 7-32, we present the point of view of *Usuario* class toward its relationship with *Persona* and *Album*. Likewise in Figure 7-33, we present the inverse side of the relationship from the *Album* class's point of view.

Figure 7-32: Usuario's point of view of the one-to-one and one-to-many relationships

```
class Usuario < ActiveRecord::Base
  self.table_name = "usuario"
  belongs_to :persona
  has_many :albums
end
```

Figure 7-33: Album point of view of the inverse relationship with Usuario - RubyOnRails

```
class Album < ActiveRecord::Base
  self.table_name = "album"
  belongs_to :usuario
end
```

We implemented the solution to the many-to-many relationship issue, with hasOne(), hasMany() and belongsTo() methods. In Figure 7-34-A and Figure 7-34-B, we present Album's and Imagen point of view of the relationship which was bridged using the AlbumXImagen class' mapping presented in Figure 7-35.

Figure 7-34: Album and Imagen point of view of the many-to-many relationship

A	`class Album < ActiveRecord::Base` `  ...  has_many :album_x_imagens end`

B	`class Imagen < ActiveRecord::Base`
	`  ...has_many :albums_x_imagens`
	`end`

Figure 7-35: AlbumXImagen bridging relationship – RubyOnRails' Active Record

```
class AlbumXImagen extends Model {
   public function Album(){
      return $this->belongsTo('App\Http\Models\Album', 'album_id', 'id');}
   public function imagen(){
      return $this->belongsTo('App\Http\Models\Album', 'imagen_id', 'id');}   }
```

7.4 DAOs and object trees

All DAOs were designed to establish a connection from the database and build an object tree as defined in section 4.2. Consequently, all of our DAOs contain three methods, one for each complexity level. In level one, we retrieve a List of *Persona*; in level two, we retrieve a list of *Usuario* including relationships with *Persona* and *Album; and in level three.* At this level, we retrieve a List of *Usuario* with full direct or indirect data, this is, with *Persona*, *Album*, *AlbumXImagen* and *Imagen* objects. As a result, this is the most complex scenario as it reads data from all tables. Below we present the way we implemented these three methods in every ORM and using pure SQL Only.

7.4.1 Hibernate

All of our Hibernate DAO methods for retrieving object trees are alike, the only code section that changes corresponds to the query *criteria*, see Figure 7-36. General *criteria* were constructed using *Session*'s *createCriteria()* method, where <*OBJ_CLASS*> is the type of the resulting data, and <*CLASS*> is the type of the *List* used to capture the Hibernate API result, generally both classes are the same.

Figure 7-36: Hibernate DAO structure in our implementation

```
// reading tree from a DAO using Hibernate
Session s = mySessionFactory.getCurrentSession();
Criteria criteria =   //instantiation
s.createCriteria(<OBJ_CLASS>);
// criteria definition
... // varies with the query level
List<CLASS> result =(List<CLASS>) criteria.list();// DB Access and object tree retrieval
```

Below we present the three criteria to query the database at the three required levels. In Figure 7-37, we show the criteria definition for the level one query; in Figure 7-38, we show the criteria that retrieve the level two object tree making sure that *Imagen or AlbumXImagen* data were not retrieved; and in Figure 7-39, we show the criteria implementation that retrieves a List of *Usuario* with full direct or indirect data, this is, with *Persona*, *Album*, *AlbumXImagen* and *Imagen* objects.

Figure 7-37: Level 1 criteria definition for Hibernate

```
createCriteria(Persona.class).add(Restrictions.like("nombre","%"+nombre+"%")).
    addOrder(Order.desc("nombre")).setMaxResults(50);
```

Figure 7-38: Level 2 criteria definition for Hibernate

```
createCriteria(Usuario.class).createAlias("persona", "p")
   .add(Restrictions.like("p.nombre", "%"+nombre+"%"))
   .addOrder(Order.desc("p.nombre")).setMaxResults(50)
   .setFetchMode("albumes", FetchMode.JOIN)
   .setFetchMode("albumes.albumXimagenList", FetchMode.SELECT);
```

Figure 7-39: Level 3 criteria definition and query forcing mechanism for Hibernate

```
criteria criteria = s.createCriteria(u.getClass())
  .createAlias("persona", "p")
  .add(Restrictions.like("p.nombre", "%" + nombre + "%"))
  .addOrder(Order.desc("p.nombre")).setMaxResults(50);
  criteria.setFetchMode("albumes", FetchMode.JOIN);

  List<Usuario> result = (List<Usuario>) criteria.list();
  // the next code forces the load of the AlbumXImagen List in each album
```

```
  // emulating, criteria.setFetchMode("albumes.albumXimagenList", FetchMode.EAGER);
              for (Usuario usuario : result) {
                     List<Album> albumes = usuario.getAlbumes();
                     for (Album album : albumes) {
                            List<AlbumXImagen> axiList = album.getAlbumXimagenList();
                            for (AlbumXImagen albumXImagen : axiList) {
                                   Imagen img = albumXImagen.getImagen();
}}}
```

The reader must note that the level three method's implementation is different from what we did in [48]. Even when we used the same case study, our previous solution used the same class annotation mapping for level one and level two queries; but to be able to execute level three, we modified that annotations. By doing this we gave birth to a second application which introduces noise when comparing the results.

Contrasting our previous work, in this implementation we forced the application to use the same annotations for the three levels and we solved the execution with another approach. The exception was caused by trying to retrieve data belonging to a relationship deep of three or more. To deal with it we simply loaded the data using multiple instructions; first, we loaded the *Album* and *Persona* data; and later, we used nested cycles to trigger the database lazy load of data by calling the getters defined for each class, see Figure 7-39.

7.4.2 MyBatis

MyBatis DAOs are quite simple. All database queries must be defined one by one, providing the SQL query to be executed and the appropriate map to translate the result in objects. This leaves the DAO with the sole task of calling those queries using the appropriate API.

We implemented those DAOs following the general template presented in Figure 7-40, where <*OBJ_CLASS*> is the type of the resulting data, and <*queryLv*> one of the three queries. To retrieve the level one object tree, we used the Persona data type and the query defined as *"selectPersons";* to retrieve the level two object tree, we used the Usuario data type and the query defined as *"lv2UserSelect"; and* to retrieve the level three object tree, we used the Usuario data type and the query defined as *"lv3UserSelect".*

Figure 7-40: Template for MyBatis DAO methods

```
@Repository public class ReadScenarioDAO {
       @Autowired private SqlSession sqlSession;
        @Transactional
```

```
        public ExperimentMeasuring <readLvlScenario>() {
                String nombre = GenericDAO.getRandomAToZChar();
                ExperimentMeasuring experiment = new ExperimentMeasuring();
                experiment.setStartingTime(new Date().getTime());
                List<OBJ_CLASS> result = sqlSession.selectList(<queryLv>, nombre);
                experiment.setEndingTime(new Date().getTime());
                return experiment; } }
```

7.4.3 Doctrine

All of our Doctrine DAO methods for retrieving object trees are alike, the only code section that changes corresponds to the query builder usage. In Figure 7-41 we present the general template we used to define DAO methods.

Figure 7-41: Template for DAO method's definition for Doctrine

```
//query builder definition
$queryBuilder-> ...//very depending on the query level
$q = $queryBuilder->getQuery();
//stops eager loadingof relationships
$q->setHint(Doctrine\ORM\Query::HINT_FORCE_PARTIAL_LOAD, true);
//get results as objects
  $models = $q->getResult();
```

Below we present the query builder to access the database at the three required levels. In Figure 7-42, we show the query builder definition for the level one query; in Figure 7-43, we show the criteria that retrieve the level two object tree making sure that *Imagen or AlbumXImagen* data were not retrieved; and in Figure 7-44, we show the level three criteria implementation that retrieves a List of *Usuario* with full direct or indirect data, this is, with *Persona*, *Album*, *AlbumXImagen* and *Imagen* objects.

Figure 7-42: Level 1 query builder definition for Doctrine

```
$queryBuilder->select('u, p, a, axi, i')
                      ->from('models\Usuario', 'u')
                      ->leftJoin('u.persona', 'p')
                      ->where('p.nombre LIKE :likeClause')
                      ->orderBy('p.nombre', 'DESC')
                      ->setFirstResult( 0 )
                      ->setMaxResults( 50 )
                      ->leftJoin('u.albumes', 'a')
                      ->leftJoin('a.albumXimagenList', 'axi')
                      ->leftJoin('axi.imagen', 'i')
                      ->setParameter('likeClause', '%' . $nombre .'%');
```

Figure 7-43: Level 2 query builder definition for Doctrine

```
$queryBuilder->select('u, p, a')
                      ->from('models\Usuario', 'u')
```

```
                        ->leftJoin('u.persona', 'p')
                        ->where('p.nombre LIKE :likeClause')
                        ->orderBy('p.nombre', 'DESC')
                        ->setFirstResult( 0 )
                        ->setMaxResults( 50 )
                        ->leftJoin('u.albumes', 'a')
                        ->setParameter('likeClause', '%' . $nombre .'%');
```

Figure 7-44: Level 3 query builder definition for Doctrine

```
$queryBuilder->select('p')
                        ->from('models\Persona', 'p')
                        ->where('p.nombre LIKE :likeClause')
                        ->orderBy('p.nombre', 'DESC')
                        ->setFirstResult( 0 )
                        ->setMaxResults( 50 )
                        ->setParameter('likeClause', '%' . $nombre .'%');
```

7.4.4 Eloquent

All of our Eloquent DAO methods for retrieving object trees are alike, the only code section that changes corresponds to the usage of the static methods inherited from the Laravel's standard Model. In Figure 7-45, we present the general template we used to define DAO methods.

Figure 7-45: DAO template for Eloquent

```
class ReadScenarioEloquentDAO{
       private $laravelDB;
       private $connection;

       public function <readLvlScenario>()
       {
              $nombre = ReadScenarioEloquentDAO::getRandomAToZChar();
              $experiment = new \App\Http\Models\ExperimentMeasuring();

              $experiment->setStartingTime(microtime(true));

              $model =  ... //very depending on the query level

              $experiment->setEndingTime(microtime(true));
              $experiment->setModels($model);
              return($experiment);
       } }
```

Below we present our implementation of those static methods to access the database at the three required levels. In Figure 7-46, we show the model retrieval for the level one query; in Figure 7-47, we show the model retrieval for the level two query, excluding *Imagen and AlbumXImagen* data; and in Figure 7-48, we show the model retrieval for the level three

query, including a List of *Usuario* with full direct or indirect data, this is, with *Persona*, *Album*, *AlbumXImagen* and *Imagen* objects.

Figure 7-46: Level 1 model retrieval for Eloquent

```
$model = Persona::where('nombre', 'like', '%'.$nombre.'%')
       ->skip(0)->take(50)->get();
```

Figure 7-47: Level 2 model retrieval for Eloquent

```
$model = \App\Http\Models\Usuario::with('persona', 'albums')//
       where('avatar', 'like', '%'.$nombre.'%')
       ->skip(0)->take(50)->get();
```

Figure 7-48: Level 2 model retrieval for Eloquent

```
$model = \App\Http\Models\Usuario
   ::with('persona', 'albums',
          'albums.albumXImagenes','albums.albumXImagenes.imagen')
       ->where('avatar', 'like', '%'.$nombre.'%')
       ->skip(0)->take(50)->get();;
```

7.4.5 RubyOnRails

Our Ruby implementation is similar to Eloquent. All of our Eloquent DAO methods for retrieving object trees are alike, the only code section that changes corresponds to the usage of the static methods inherited from the RubyOnRails' *ActiveRecord::Base*. In Figure 7-49, we present the general template we used to define DAO methods.

Figure 7-49: Template for DAO method's definition for RubyOnRails

```
class ReadScenarioRailsDAO
  def <readlv>
    @experiment = ExperimentMeasuring.new()
    @experiment.setStartingTime(Time.now)
    @char = randChar
    @models= ... //very depending on the query level
    @experiment.setEndigTime(Time.now)
    @experiment.setModels(@models)
    return  @experiment
  end
```

Below we present our implementation of those static methods to access the database at the three required levels. In Figure 7-50, we show the model retrieval for the level one query; in Figure 7-51, we show model retrieval for the level two query, excluding *Imagen and AlbumXImagen* data; and in Figure 7-52, we show model retrieval for the level three

query, including a List of *Usuario* with full direct or indirect data, this is, with *Persona*, *Album*, *AlbumXImagen* and *Imagen* objects.

Figure 7-50: Level 1 model retrieval for RubyOnRails' Active Record

```
@models= Persona.where("nombre LIKE :char", {:char =>'%' + @char + '%' })
                .order('nombre DESC').limit(50).offset(0)
```

Figure 7-51: Level 2 model retrieval for RubyOnRails' Active Record

```
@models= Usuario.joins(:persona)
                .where('persona.nombre LIKE :char', {:char =>'%' + @char + '%'})
                .order('persona.nombre DESC').limit(50).offset(0)
                .joins(:albums)
                .eager_load(:persona, :albums)
```

Figure 7-52: Level 3 model retrieval for RubyOnRails' Active Record

```
@models= Usuario.joins(:persona)
                .where('persona.nombre LIKE :char', {:char =>'%' + @char + '%'})
                 .order('persona.nombre DESC').limit(50).offset(0)
                 .joins(:albums)
                 .joins(:albums => :album_x_imagens)
                 .joins(:albums => {:album_x_imagens => :imagen})
                 .eager_load(:persona, :albums, :albums => :album_x_imagens, :albums =>
                            {:album_x_imagens => :imagen}  )
```

7.4.6 Pure SQL DAO

To be able to compare SQL response times with those from ORM implementations, we need to recreate the same conditions. Given that in the pure SQL approach, *executeQuery()* method responds with a *ResultSet* collection rather than the object trees retrieved by every off-the-shelf ORM in this study, we need to transform that *resultSet*. To be able to recreate the same conditions, we need to define a way to deal with the Resultset-Object translation.

We propose to deal with that translation Object assemble using typed maps. This strategy divides every DAO method in two major sections. In the first section, we accessed the database using a single query; and in the second section, we built the required object tree.

Instantiating objects from the Java ResultSet matrix, is a process that needs special care. Transforming tuples into objects is a simple task, when the resulting data belong to a single class (like in level 1 scenario); however, the same task becomes harder when such data

belong to many classes (like in level two and three). As an effort to optimize this process and reduce the time the program takes to transform level two and three ResultSets into objects, we used a typed map - HashMap<id, object>- for every row contained in the ResultSet. This mechanism has the advantage of reading every tuple just one time. Meaning that, this approach solves the problem with and algorithm with order O(n).

This approach increased the reliability of the study as it balances the requirements and the challenges. Our solution avoids the use of an ORM but at the same time achieves the same object tree. Moreover, the process of building the trees followed the best practices and it was done using an algorithm with a satisfactory behavior.

Below we present our SQL sentences and the data assemble process. In Figure 7-53, we show the tree retrieval for the level one query; as the implementation is long, in Appendix F, we show the tree retrieval for the level two query, excluding *Imagen and AlbumXImagen* data; and for the same reason, in appendix G, we show the tree retrieval for the level three query, including a List of *Usuario* with full direct or indirect data, this is, with *Persona*, *Album*, *AlbumXImagen* and *Imagen* objects.

Figure 7-53: Level 1 DAO method for pure SQL implementation

```
Connection connection = myDataSource.getConnection();
Statement statement = connection.createStatement();
ResultSet result=
  statement.executeQuery("select * from persona WHERE nombre like '%" + nombre + "%'"
    + " ORDER BY nombre DESC "
    + " LIMIT 0, " + EXPERIMENT_STANDART_PAGE_SIZE);

  List<Persona> personas = new ArrayList<Persona>();
  while(result.next()) {
        Persona pAux = new Persona();

        String rowID = result.getString("id");
        pAux.setId(Long.parseLong(rowID));

        String rowName = result.getString("nombre");
        pAux.setNombre(rowName);
        String rowCedula = result.getString("cedula");
        pAux.setCedula(rowCedula);

        personas.add(pAux); // build list
}
```

8 Conclusion and future work

8.1 About our WAF-ORM Evaluation Framework

In this research we proposed a WAF-ORM evaluation framework to assess the tradeoff between maintainability and performance, and we applied it to five WAF-ORM technologies. We analyzed how overhead varies with the complexity of the query in a complex and flexible case study susceptible to be implemented in other technologies.

We tested those queries maintaining the same conditions for the three query complexity levels and database sizes. To that end, we ran all tests the same day, on the same machine, with the same process load, using default configurations both in Windows and Linux, and maintaining the same database optimization features. This way, we left the complexity of the query as the only variable responsible for the query performance comparison.

Our main conclusion is that with exception of Rails' Active Record, performance is similar to non-optimized SQL queries. This is, when queries need to access a single table, response times are comparable to those from SQL and even better (as reported in [45]). On the contrary, when the query complexity grows Hibernate proved to be much slower than optimized SQL queries in windows. This happens because Hibernate cannot find the optimal join execution order, when accessing multiple tables with a relationship-deep of two levels or more. However, this gap is less severe in Linux where MariaDB benefits the performance of Hibernate, Doctrine, and Mybatis but makes slower our solution of manually coded SQL queries.

We want to highlight the fact that query complexity was the most important variable in our study, and it proved to be the reason of the rapidly growing response times in Windows. Moreover, the performance gap related to Hibernate cannot be easily avoided by controlling joins or sub-queries execution order. An ideal ORM should provide a simple but powerful mechanism to control, when necessary, how those queries must access database tables

and ease the task of querying them with a single sentence. We also found that Linux is more efficient when handling larger databases, as a consequence, more research is needed to prove that this behavior will be constant when the database grows even more. For these reasons query complexity was an important variable to consider when evaluating performance in ORM tools.

Also, performance proved to be sensible to the platform where tests are performed. Windows and Linux show produced different response times for the same query. Also, the performance gap between ORM's and SQL's queries was more severe on Windows.

This means that novice developers need to be careful when using ORM tools in complex scenarios to avoid overhead issues. In those cases, ORM usage will benefit developers only in the maintainability aspect. We also noted that in very complex scenarios, Hibernate will even lose the query optimization features available in different database products as it exhibits the n+1 queries problem. The same problem could explain Rails' Active Record, but further research is needed to prove it. Another overhead issue we found is that some ORM documentation points out that eagerly loading address the n+1 queries problem and easies the query definition however, after testing this resulted in even more queries being delivered to the database. It seems that eagerly loading is sensible to the complexity of the joins used in the query. Therefore, it is recommended that novice developers extensively test this type of queries before deploying them.

Finally, we want to point out that we modified the case study presented in [48] to avoid changing our class model to be able to retrieve data in level 3 scenario when using Hibernate. In [48], this modification altered the semantics of other queries, this is, by making eager the load of two attributes in level three complexity level, authors also forced level two and level one to load those attributes when they are not required. We solved this issue in our Linux implementation, by manually forcing Hibernate's lazy load of attributes using a cycle based script.

8.2 About Maintainability

Under the optics provided by this research, our maintainability metric aids in the measuring how much the logic in the persistence layer is bound to the database structure. This is a

valuable contribution because it helps to analyze the strength of the abstraction barrier achieved by the persistence layer using a particular ORM. The reader must note that the persistence layer is intended to bring independence between the modeling of database and the object oriented program. Noting this helps developers to be aware that some conventions we found in modern ORMs are aimed to gain performance but lead to weak abstraction barriers. As changes propagate easily trough the program's layers, weak abstraction barriers could lead to software harder and more expensive to both maintain and extend.

Under this line of thinking, none of the analyzed ORM allows the object oriented program to be really independent of the database structure, threatening maintainability. The binding between the database and the object oriented program is present because they focus development around the relational model. Even when they claim to be object oriented tools, they give little importance to the object oriented design. Arango [30] points out that the business classes are the basic element of software modularization, and, any other form of modularization must respect the boundaries of such classes. This idea is left behind by ORMs like Eloquent where business classes are empty PHP classes that aim to reduce development time by loading as attribute anything stored in the linked table.

As a consequence maintainability is threatened by any change in the database, even by those as simple as column renaming. Views, controllers and other subprograms consuming the classes defined by nowadays WAFs and ORMs are in fact aware of the column names and database structure. As a consequence, those business classes are poor abstraction barriers. The reader must note that when classes have such abstract definition, they lose the benefits of better data and process encapsulation as proposed by authors from [25]. Moreover they are less effective at capturing models of the real world as proposed by authors from [5]. This means that those business classes are a consequence of software development rather than the guiding thread useful to define a common language to build all software components.

8.3 About the tradeoff between maintainability and performance

We found no clear evidence of tradeoff between maintainability and performance. The reader must remember that none of the ORMs analyzed truly achieve that the object oriented application becomes oblivious of the database structure. Additionally, our later results in Linux show that the response time gap between ORMs and SQL queries is not big. This is, ORMs with both strong and weak abstraction level perform close to SQL queries and, in some cases, even better.

Under this line of thinking, our framework proposes a useful way of analyzing performance and maintainability simultaneously that provided confidence about the possibility of getting the best of both worlds. This idea is also grounded on the facts that on one side, ORMs in this study uses the same data as input to craft SQL queries, the difference in maintainability arises because some ORMs take those input data from every query and some of them take them from metadata that severs a unified set of mapping rules.

Therefore, we can conclude that maintainability is orthogonal to efficiency.

8.4 Future Work

As future work, we will study more in depth the causes that lead Hibernate and Rails' Active record to poor performance. Likewise, we will study the causes of the surprisingly fast response times of Mybatis and Eloquent. To achieve this, we will continue the development of the WAFs Learning wiki [57] maintained by our research group, and we will support undergraduate thesis in this field. Additionally, we will extend the analysis over ORM tools to explain in a more accurate way the unexpected behavior we found. Finally, we will evaluate other ORM tools. To do so, we will use our case study as a benchmark.

A. Apendix: Spring *pom.xml* file for Hibernate and MySQL

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
         <modelVersion>4.0.0</modelVersion>
         <groupId>co.edu</groupId>
         <artifactId>perf</artifactId>
         <name>perf</name>
         <packaging>war</packaging>
         <version>1.0.0-BUILD-SNAPSHOT</version>
         <properties>
                  <java-version>1.6</java-version>
                  <org.springframework-version>3.1.1.RELEASE</org.springframework-version>
                  <org.aspectj-version>1.6.10</org.aspectj-version>
                  <org.slf4j-version>1.6.6</org.slf4j-version>
         </properties>
         <dependencies>
                  <!-- Spring -->
                  <dependency>
                           <groupId>org.springframework</groupId>
                           <artifactId>spring-context</artifactId>
                           <version>${org.springframework-version}</version>
                           <exclusions>
                                    <!-- Exclude Commons Logging in favor of SLF4j -->
                                    <exclusion>
                                             <groupId>commons-logging</groupId>
                                             <artifactId>commons-logging</artifactId>
                                     </exclusion>
                           </exclusions>
                  </dependency>
                  <dependency>
                           <groupId>org.springframework</groupId>
                           <artifactId>spring-webmvc</artifactId>
                           <version>${org.springframework-version}</version>
                  </dependency>

                  <!-- AspectJ -->
                  <dependency>
                           <groupId>org.aspectj</groupId>
                           <artifactId>aspectjrt</artifactId>
                           <version>${org.aspectj-version}</version>
                  </dependency>

                  <!-- Logging -->
                  <dependency>
                           <groupId>org.slf4j</groupId>
                           <artifactId>slf4j-api</artifactId>
                           <version>${org.slf4j-version}</version>
                  </dependency>
                  <dependency>
                           <groupId>org.slf4j</groupId>
                           <artifactId>jcl-over-slf4j</artifactId>
                           <version>${org.slf4j-version}</version>
                           <scope>runtime</scope>
```

```
		</dependency>
		<dependency>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-log4j12</artifactId>
			<version>${org.slf4j-version}</version>
			<exclusions>
				<exclusion>
					<groupId>javax.mail</groupId>
					<artifactId>mail</artifactId>
				</exclusion>
				<exclusion>
					<groupId>javax.jms</groupId>
					<artifactId>jms</artifactId>
				</exclusion>
				<exclusion>
					<groupId>com.sun.jdmk</groupId>
					<artifactId>jmxtools</artifactId>
				</exclusion>
				<exclusion>
					<groupId>com.sun.jmx</groupId>
					<artifactId>jmxri</artifactId>
				</exclusion>
			</exclusions>
			<scope>runtime</scope>
		</dependency>

		<!-- @Inject -->
		<dependency>
			<groupId>javax.inject</groupId>
			<artifactId>javax.inject</artifactId>
			<version>1</version>
		</dependency>

		<!-- Servlet -->
		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>servlet-api</artifactId>
			<version>2.5</version>
			<scope>provided</scope>
		</dependency>
		<dependency>
			<groupId>javax.servlet.jsp</groupId>
			<artifactId>jsp-api</artifactId>
			<version>2.1</version>
			<scope>provided</scope>
		</dependency>
		<!-- Test -->
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>4.7</version>
			<scope>test</scope>
		</dependency>

		<!-- MySQL database driver -->
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<version>5.1.10</version>
		</dependency>

		<!-- DEPENDENCIAS ADICIONALES PARA EL MANEJO DE ENTIDADES  -->
  <!-- Hibernate -->
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-core</artifactId>
			<version>4.2.2.Final</version>
		</dependency>

  <dependency>
<groupId>org.springframework</groupId>
```

```
        <artifactId>spring-orm</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>

  <dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>

  <!-- Others -->
          <dependency>
              <groupId>commons-dbcp</groupId>
              <artifactId>commons-dbcp</artifactId>
              <version>1.4</version>
          </dependency>
    <!-- AOP dependency -->
    <dependency>
          <groupId>cglib</groupId>
          <artifactId>cglib</artifactId>
          <version>2.2</version>
    </dependency>

          </dependencies>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-eclipse-plugin</artifactId>
                <version>2.9</version>
                <configuration>
                    <additionalProjectnatures>

<projectnature>org.springframework.ide.eclipse.core.springnature</projectnature>
                    </additionalProjectnatures>
                    <additionalBuildcommands>

<buildcommand>org.springframework.ide.eclipse.core.springbuilder</buildcommand>
                    </additionalBuildcommands>
                    <downloadSources>true</downloadSources>
                    <downloadJavadocs>true</downloadJavadocs>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.5.1</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                    <compilerArgument>-Xlint:all</compilerArgument>
                    <showWarnings>true</showWarnings>
                    <showDeprecation>true</showDeprecation>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
                <configuration>
                    <mainClass>org.test.int1.Main</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

B. Appendix: Spring servlet-context.xml file for Hibernate and MySQL

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

          <!-- Enables the Spring MVC @Controller programming model -->
          <mvc:annotation-driven />

          <!-- Handles HTTP GET requests for /resources/** by efficiently serving
                    up static resources in the ${webappRoot}/resources directory -->
          <mvc:resources mapping="/resources/**" location="/resources/" />

          <!-- Resolves views selected for rendering by @Controllers to .jsp resources
                    in the /WEB-INF/views directory -->
          <beans:bean

          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                    <beans:property name="prefix" value="/WEB-INF/views/" />
                    <beans:property name="suffix" value=".jsp" />
          </beans:bean>

          <context:component-scan base-package="co.edu.perf" />

          <!-- JDBC Data Source. It is assumed you have MySQL running on localhost
          port 3306 with username root and blank password. Change below if it's not
          the case -->
          <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
                    destroy-method="close">
                    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                    <property name="url" value="jdbc:mysql://localhost:3306/performance" />
                    <property name="username" value="root" />
                    <property name="password" value="" />
                    <property name="validationQuery" value="SELECT 1" />
          </bean>

          <!-- Hibernate Session Factory -->
          <bean id="mySessionFactory"
                    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
                    <property name="dataSource" ref="myDataSource" />
```

```
                <property name="packagesToScan">
                        <array>
                                <value>co.edu.perf</value>
                        </array>
                </property>
                <property name="hibernateProperties">
                        <props>
                                <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                                <prop key="hibernate.max_fetch_depth">3</prop>
                                <prop key="hibernate.jdbc.fetch_size">50</prop>
                                <prop key="hibernate.jdbc.batch_size">10</prop>
                                <prop key="hibernate.show_sql">true</prop>
                        </props>
                </property>
        </bean>
        <!-- Hibernate Transaction Manager -->
        <bean id="transactionManager"
                class="org.springframework.orm.hibernate4.HibernateTransactionManager">
                <property name="sessionFactory" ref="mySessionFactory" />
        </bean>
        <!-- Activates annotation based transaction management -->
        <tx:annotation-driven transaction-manager="transactionManager" />

        <bean id="genericDAO" class="co.edu.perf.manager.GenericDAO" />
        <bean id="dataBasePupulationDAO"
class="co.edu.perf.manager.DataBasePopulationDAO" />
</beans>
```

C. Appendix: Spring pom.xml file for MyBatis

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>co.unal</groupId>
        <artifactId>perfimybatis</artifactId>
        <name>perfmybatis</name>
        <packaging>war</packaging>
        <version>1.0.0-BUILD-SNAPSHOT</version>
        <properties>
                <java-version>1.6</java-version>
                <org.springframework-version>3.1.1.RELEASE</org.springframework-version>
                <org.aspectj-version>1.6.10</org.aspectj-version>
                <org.slf4j-version>1.6.6</org.slf4j-version>
        </properties>
        <dependencies>
                <!-- Spring -->
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>${org.springframework-version}</version>
                        <exclusions>
                                <!-- Exclude Commons Logging in favor of SLF4j -->
                                <exclusion>
                                        <groupId>commons-logging</groupId>
                                        <artifactId>commons-logging</artifactId>
                                 </exclusion>
                        </exclusions>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webmvc</artifactId>
                        <version>${org.springframework-version}</version>
                </dependency>
                <!-- AspectJ -->
                <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjrt</artifactId>
                        <version>${org.aspectj-version}</version>
                </dependency>
                <!-- Logging -->
                <dependency>
                        <groupId>org.slf4j</groupId>
                        <artifactId>slf4j-api</artifactId>
                        <version>${org.slf4j-version}</version>
                </dependency>
                <dependency>
                        <groupId>org.slf4j</groupId>
                        <artifactId>jcl-over-slf4j</artifactId>
                        <version>${org.slf4j-version}</version>
```

```
		<scope>runtime</scope>
	</dependency>
	<dependency>
		<groupId>org.slf4j</groupId>
		<artifactId>slf4j-log4j12</artifactId>
		<version>${org.slf4j-version}</version>
		<scope>runtime</scope>
	</dependency>
	<dependency>
		<groupId>log4j</groupId>
		<artifactId>log4j</artifactId>
		<version>1.2.15</version>
		<exclusions>
			<exclusion>
				<groupId>javax.mail</groupId>
				<artifactId>mail</artifactId>
			</exclusion>
			<exclusion>
				<groupId>javax.jms</groupId>
				<artifactId>jms</artifactId>
			</exclusion>
			<exclusion>
				<groupId>com.sun.jdmk</groupId>
				<artifactId>jmxtools</artifactId>
			</exclusion>
			<exclusion>
				<groupId>com.sun.jmx</groupId>
				<artifactId>jmxri</artifactId>
			</exclusion>
		</exclusions>
		<scope>runtime</scope>
	</dependency>
	<!-- @Inject -->
	<dependency>
		<groupId>javax.inject</groupId>
		<artifactId>javax.inject</artifactId>
		<version>1</version>
	</dependency>
	<!-- Servlet -->
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>servlet-api</artifactId>
		<version>2.5</version>
		<scope>provided</scope>
	</dependency>
	<dependency>
		<groupId>javax.servlet.jsp</groupId>
		<artifactId>jsp-api</artifactId>
		<version>2.1</version>
		<scope>provided</scope>
	</dependency>
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>jstl</artifactId>
		<version>1.2</version>
	</dependency>
	<!-- Test -->
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.7</version>
		<scope>test</scope>
	</dependency>

	<!-- DEPENDENCIAS PARA MYBATIS -->
	<dependency>
		<groupId>org.mybatis</groupId>
		<artifactId>mybatis</artifactId>
		<version>3.2.8</version>
	</dependency>
```

```
                <dependency>
                    <groupId>org.mybatis</groupId>
                    <artifactId>mybatis-spring</artifactId>
                    <version>1.1.1</version>
                </dependency>

                <!-- MySQL database driver -->
                <dependency>
                          <groupId>mysql</groupId>
                          <artifactId>mysql-connector-java</artifactId>
                          <version>5.1.10</version>
                </dependency>
                <!-- Others -->
                <dependency>
                    <groupId>commons-dbcp</groupId>
                    <artifactId>commons-dbcp</artifactId>
                    <version>1.4</version>
                </dependency>
                 <dependency>
           <groupId>commons-logging</groupId>
           <artifactId>commons-logging</artifactId>
           <version>1.1.1</version>
       </dependency>
                <!-- AOP dependency -->
             <dependency>
                   <groupId>cglib</groupId>
                          <artifactId>cglib</artifactId>
                          <version>2.2</version>
             </dependency>
         </dependencies>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-eclipse-plugin</artifactId>
                <version>2.9</version>
                <configuration>
                    <additionalProjectnatures>

<projectnature>org.springframework.ide.eclipse.core.springnature</projectnature>
                    </additionalProjectnatures>
                    <additionalBuildcommands>

<buildcommand>org.springframework.ide.eclipse.core.springbuilder</buildcommand>
                    </additionalBuildcommands>
                    <downloadSources>true</downloadSources>
                    <downloadJavadocs>true</downloadJavadocs>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.5.1</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                    <compilerArgument>-Xlint:all</compilerArgument>
                    <showWarnings>true</showWarnings>
                    <showDeprecation>true</showDeprecation>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
                <configuration>
                    <mainClass>org.test.int1.Main</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

D. Appendix: Spring servlet-context.xml file for MyBatis

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
<!-- DispatcherServlet Context: defines this servlet's request-processing infrastructure --
>
          <!-- Enables the Spring MVC @Controller programming model -->
          <mvc:annotation-driven />

<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources
in the ${webappRoot}/resources directory -->
          <mvc:resources mapping="/resources/**" location="/resources/" />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-
INF/views directory -->
          <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                    <beans:property name="prefix" value="/WEB-INF/views/" />
                    <beans:property name="suffix" value=".jsp" />
          </beans:bean>

          <context:component-scan base-package="co.unal.perfimybatis" />

          <!-- PARA LEER LAS PROPIEDADES DESDE EL  config.properties-->
         <util:properties id="nodeProperty" location="classpath:config/config.properties" />
          <context:property-placeholder properties-ref="nodeProperty" />

          <!-- BEANS DE MYBATIS -->
          <bean id='dataSource' class='org.apache.commons.dbcp.BasicDataSource'>
                    <property name='driverClassName' value='${jdbc.driverClassName}'/>
                    <property name='url' value='${jdbc.url}'/>
                    <property name='username' value='${jdbc.username}'/>
                    <property name='password' value='${jdbc.password}'/>
          </bean>

          <!-- Para obtener el mybatis sessionFactory -->
          <bean id='sqlSessionFactory' class='org.mybatis.spring.SqlSessionFactoryBean'>
          <property name='dataSource' ref='dataSource' />
```

```
        <property name='typeAliasesPackage' value='co.unal.perfimybatis.entity'/>
        <property name='mapperLocations' value='classpath*:mapper/*.xml' />
        </bean>

        <!-- Configure SqlSessionTemplate which provides ThreadSafe SqlSession object. --
>
        <bean id='sqlSession' class='org.mybatis.spring.SqlSessionTemplate'>
                  <constructor-arg index='0' ref='sqlSessionFactory' />
        </bean>

        <!-- To be able to inject Mappers directly we should register
         org.mybatis.spring.mapper.MapperScannerConfigurer and configure the package name
         where to find Mapper Interfaces. -->

        <tx:annotation-driven transaction-manager="transactionManager"/>
        <bean id='transactionManager'
              class='org.springframework.jdbc.datasource.DataSourceTransactionManager'>
        <property name='dataSource' ref='dataSource' />
        </bean>
</beans>
```

E. Appendix: Doctrine.php loader for CodeIgniter

```
<?php
use Doctrine\Common\ClassLoader,
Doctrine\ORM\Configuration,
Doctrine\ORM\EntityManager,
Doctrine\Common\Cache\ArrayCache,
Doctrine\DBAL\Logging\EchoSQLLogger;

class Doctrine {
          public $em = null;
          public function __construct()
          {
                    // load database configuration from CodeIgniter
                    require_once APPPATH.'config/database.php';
// Set up class loading. You could use different autoloaders, provided by your favorite
//framework, if you want to.
                    require_once APPPATH.'libraries/Doctrine/Common/ClassLoader.php';

                    $doctrineClassLoader = new ClassLoader('Doctrine',
APPPATH.'libraries');
                    $doctrineClassLoader->register();
//                  $temp=rtrim(APPPATH, "/" );die($temp);
                    $entitiesClassLoader = new ClassLoader('models', rtrim(APPPATH, "/" ));
                    $entitiesClassLoader->setFileExtension('.php');
                    $entitiesClassLoader->register();
                    $proxiesClassLoader = new ClassLoader('Proxies',
APPPATH.'models/proxies');
                    $proxiesClassLoader->register();

                    // Set up caches
                    $config = new Configuration;
                    $cache = new ArrayCache;
                    $config->setMetadataCacheImpl($cache);
                    $driverImpl = $config-
>newDefaultAnnotationDriver(array(APPPATH.'models/Entities'));
//                  $driverImpl = $config-
>newDefaultAnnotationDriver(array(APPPATH.'models'));
                    $config->setMetadataDriverImpl($driverImpl);
                    $config->setQueryCacheImpl($cache);

                    $config->setQueryCacheImpl($cache);

                    // Proxy configuration
                    $config->setProxyDir(APPPATH.'/models/proxies');
                    $config->setProxyNamespace('Proxies');

                    // Set up logger
                    $logger = new EchoSQLLogger;
                    $config->setSQLLogger($logger);

                    $config->setAutoGenerateProxyClasses( TRUE );

                    // Database connection information
```

```
                $connectionOptions = array(
                                    'driver' => 'pdo_mysql',
                                    'user' =>     $db['default']['username'],
                                    'password' => $db['default']['password'],
                                    'host' =>     $db['default']['hostname'],
                                    'dbname' =>   $db['default']['database']
                );

                // Create EntityManager
                $this->em = EntityManager::create($connectionOptions, $config);
        }
}
```

F. Appendix: Level 2 DAO method for pure SQL implementation with typed maps

```
Connection connection = myDataSource.getConnection();

Statement statement = connection.createStatement();

ResultSet result= statement. executeQuery(

    "SELECT uxp.person_id, uxp.person_nombre, uxp.person_cedula,"

    + " uxp.id, uxp.avatar, uxp.nick_name, "

    + " al.id AS album_id, al.nombre AS album_nombre, al.descripcion "

    + " FROM ((  SELECT u.id, u.avatar, u.nick_name, "

        + " p.id AS person_id, p.nombre AS person_nombre, p.cedula as person_cedula "

        + "FROM usuario u LEFT JOIN persona p ON p.id = u.persona_id "

        + " WHERE p.nombre like '%" + nombre + "%'  "

        + " ORDER BY p.nombre DESC LIMIT 0, 50) uxp)"

    + " LEFT JOIN album al ON al.usuario_id = uxp.id ");


//build a hash map to assemble Usuario and Persona,

Map<String, Usuario> map = new HashMap<String, Usuario>();

List<Usuario> usuarios = new ArrayList<Usuario>();


while(result.next()) {

        String rowID = result.getString("id");

        if(!map.containsKey(rowID)) {

                // If a user is not mapped, we add him to the map
```

```
                Persona persona = new Persona();
                persona.setId(Long.parseLong(result.getString("person_id")));
                persona.setNombre(result.getString("person_nombre"));
                persona.setCedula(result.getString("person_cedula"));
                Usuario usuario = new Usuario();
                usuario.setPersona(persona);
                persona.setUsuario(usuario);
                usuario.setId(Long.parseLong(rowID));
                usuario.setNickName(result.getString("nick_name"));
                usuario.setAvatar(result.getString("avatar"));
                usuario.setAlbumes(new ArrayList<Album>());
                usuarios.add(usuario);
                map.put(rowID, usuario);
        }

        // User already in the map, we add the Album
        Album albumTemp = new Album();
        albumTemp.setId(Long.parseLong(result.getString("album_id")));
        albumTemp.setNombre(result.getString("album_nombre"));
        albumTemp.setDescripcion(result.getString("descripcion"));
        albumTemp.setUsuario(map.get(rowID));
        map.get(rowID).getAlbumes().add(albumTemp);
 }
}}
```

G. Appendix: Level 3 DAO method for pure SQL implementation with typed maps

```
Connection connection = myDataSource.getConnection();
   Statement statement = connection.createStatement();
   ResultSet result= statement.
   executeQuery("SELECT uxp.person_id, uxp.person_nombre, uxp.person_cedula, "
   + " uxp.id, uxp.avatar, uxp.nick_name, "
   + " al.id AS album_id, al.nombre AS album_nombre,al.descripcion as album_descripcion,
"
   + " axi.numero_orden, "
   + " i.id as imagen_id, i.foto as imagen_foto, i.descripcion as imagen_descripcion, "
   + " i.titulo as imagen_titulo, i.comentarios as imagen_comentarios "
   + " FROM ((  SELECT u.id, u.avatar, u.nick_name, "
        + " p.id AS person_id, p.nombre AS person_nombre, p.cedula as person_cedula "
        + " FROM usuario u "
        + " LEFT JOIN persona p ON p.id = u.persona_id "
        + " WHERE p.nombre like '%" + nombre + "%'  "
        + " ORDER BY p.nombre DESC LIMIT 0, 50) uxp)"
   + " LEFT JOIN album al ON al.usuario_id = uxp.id "
   + " LEFT JOIN album_x_imagen axi ON al.id = axi.album_id "
   + " LEFT JOIN imagen i on axi.imagen_id = i.id ");

List<Usuario> usuarios = new ArrayList<Usuario>();
```

```
//hash map to build  usuario and album
Map<String, Usuario> map = new HashMap<String, Usuario>();
Map<String, Album> mapAlbum = new HashMap<String, Album>();
while(result.next()) {
        String rowID = result.getString("id");
        String albumId = result.getString("album_id");
        String imagenId = result.getString("imagen_id");

        if(!map.containsKey(rowID)) {
        // if a usuario is not in the map, we add him
        Persona persona = new Persona();
        persona.setId(Long.parseLong(result.getString("person_id")));
        persona.setNombre(result.getString("person_nombre"));
        persona.setCedula(result.getString("person_cedula"));
        Usuario usuario = new Usuario();
        usuario.setPersona(persona);
        persona.setUsuario(usuario);
        usuario.setId(Long.parseLong(rowID));
        usuario.setNickName(result.getString("nick_name"));
        usuario.setAvatar(result.getString("avatar"));
        usuario.setAlbumes(new ArrayList<Album>());
        usuarios.add(usuario);
        map.put(rowID, usuario);
}

// Usuario already on the ma, we add the Album
if( (albumId != null) && !(mapAlbum.containsKey(albumId)) ) {
        Album albumTemp = new Album();
        albumTemp.setId(Long.parseLong(albumId));
        albumTemp.setNombre(result.getString("album_nombre"));
```

```
        albumTemp.setDescripcion(result.getString("album_descripcion"));
        albumTemp.setAlbumXimagenList(new ArrayList<AlbumXImagen>());

        //add album to the corresponding Usuario
        map.get(rowID).getAlbumes().add(albumTemp);

        //add album to the map
        mapAlbum.put(albumId, albumTemp);
}

//to this point Album already exist. So, we add the Imagen
if(imagenId != null) {

        // we créate the image
        Imagen img = new Imagen();
        img.setId(Long.parseLong(imagenId));
        img.setFoto(result.getString("imagen_foto"));
        img.setTitulo(result.getString("imagen_titulo"));
        img.setDescripcion(result.getString("imagen_descripcion"));
        img.setComentarios(result.getString("imagen_comentarios"));

        // create album_x_imagen
        AlbumXImagen axi = new AlbumXImagen();
        axi.setNumeroOrden(result.getString("numero_orden"));
        axi.setAlbum(mapAlbum.get(albumId));
        axi.setImagen(img);

        //add album_x_imagen to its album
        mapAlbum.get(albumId).getAlbumXimagenList().add(axi);
}
}}
```

Bibliography

[1] D. Schwabe, G. Rossi, L. Esmeraldo, and F. Lyardet, “Web Design Frameworks: An Approach to Improve Reuse in Web Applications,” vol. 2016, S. Murugesan and Y. Deshpande, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[2] R. E. Johnson, “Components, frameworks, patterns,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 3, pp. 10–17, May 1997.

[3] G. E. Krasner and S. T. Pope, “A description of the model-view-controller user interface paradigm in the smalltalk-80 system,” *J. object oriented Program.*, vol. 1, pp. 26–49, 1988.

[4] A. Leff and J. T. Rayfield, “Web-application development using the Model/View/Controller design pattern,” in *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, 2001, pp. 118–127.

[5] G. Booch, “Object-oriented development,” *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, pp. 211–221, Feb. 1986.

[6] C. Murdaca, “An Object-Relational Compiler,” in *2009 WRI World Congress on Computer Science and Information Engineering*, 2009, vol. 7, pp. 438–442.

[7] Oracle, “MySQL,” 1995. [Online]. Available: https://www.mysql.com/. [Accessed: 08-Mar-2016].

[8] M. Widenius, “MariaDB.org - Continuity and open collaboration,” 2009. [Online]. Available: https://mariadb.org/. [Accessed: 08-Mar-2016].

[9] M. Stonebraker, “PostgreSQL: The world’s most advanced open source database,” 1995. [Online]. Available: http://www.postgresql.org/. [Accessed: 08-Mar-2016].

[10] S. Software Development Laboratories, “Oracle Database,” 1977. [Online]. Available: https://www.oracle.com/database/index.html. [Accessed: 08-Mar-2016].

[11] Microsoft, “SQL Server Database,” 1989. [Online]. Available: https://www.microsoft.com/web/platform/database.aspx. [Accessed: 08-Mar-2016].

[12] C. Ireland, D. Bowers, M. Newton, and K. Waugh, “A Classification of Object-Relational Impedance Mismatch,” in *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*, 2009, pp. 36–43.

[13] J. Yoder, R. Johnson, and Q. Wilson, “Connecting business objects to relational databases,” *Urbana*, 2005.

[14] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, Second Edi. Cambridge, United States: The MIT Press, 1996.

[15] Hibernate, “Hibernate. Everything data.” [Online]. Available: http://hibernate.org/. [Accessed: 10-Mar-2016].

[16] MyBatis, “MyBatis.” [Online]. Available: http://www.mybatis.org/mybatis-3/. [Accessed: 10-Mar-2016].

[17] T. Doctrine, “Doctrine Project,” 2006. [Online]. Available: http://www.doctrine-project.org/. [Accessed: 10-Mar-2016].

[18] D. Heinemeier Hansson, “Ruby on Rails,” 2005. [Online]. Available: http://rubyonrails.org/. [Accessed: 10-Mar-2016].

[19] T. Otwell, “Laravel - The PHP Framework For Web Artisans,” 2011. [Online]. Available: https://laravel.com/. [Accessed: 10-Mar-2016].

[20] M. Björemo and P. Trninić, “Evaluation of web application frameworks-Evaluation of web application frameworks with regards to rapid development.,” 2010.

[21] J. Plekhanova, “Evaluating web development frameworks: Django, Ruby on Rails and CakePHP,” *Inst. Bus. Inf. Technol.*, 2009.

[22] P. Wang, “Comparison of Four Popular Java Web Framework Implementations: Struts1. X, WebWork2. 2X, Tapestry4, JSF1. 2,” 2008.

[23] M. Canales, “A Comparative Study of Rapid Development Frameworks for the Creation of a Language Placement Exam Template,” 2010.

[24] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, “Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks,” *IEEE Trans. Softw. Eng.*, vol. PP, no. 99, pp. 1–1, 2016.

[25] N. Wilde and R. Huitt, “Maintenance support for object-oriented programs,” *IEEE Trans. Softw. Eng.*, vol. 18, no. 12, pp. 1038–1044, Dec. 1992.

[26] R. W. Selby, “Enabling reuse-based software development of large-scale systems,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 495–510, Jun. 2005.

[27] D. Mancl and W. Havanas, “A study of the impact of C++ on software maintenance,” in *Proceedings. Conference on Software Maintenance 1990*, 1990, pp. 63–69.

[28] G. Low and S. Huan, “Impact of object oriented development on software quality,” in *STEP ’99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, 1999, pp. 3–11.

[29] H. Ran, W. Zhuo, and X. Jianfeng, “Web Quality of Agile Web Development,” *2009 IITA Int. Conf. Serv. Sci. Manag. Eng.*, pp. 426–429, Jul. 2009.

[30] F. Arango Isaza, *Ingeniería de Software un Libro WEB*. 2006.

[31] J. Kienzle and A. Romanovsky, “Framework based on design patterns for providing persistence in object-oriented programming languages,” *IEE Proc. - Softw.*, vol. 149, no. 3, p. 77, 2002.

[32] E. Codd, “A relational model of data for large shared data banks,” *Commun. ACM*, 1970.

[33] S. Philippi, “Model driven generation and testing of object-relational mappings,” *J. Syst. Softw.*, 2005.

[34] B. Kitchenham, “Procedures for performing systematic reviews,” *Keele, UK, Keele Univ.*, 2004.

[35] B. Kitchenham and O. Brereton, “Systematic literature reviews in software engineering–a systematic literature review,” *Inf. Softw. …*, 2009.

[36] P. Carbonnelle, “PYPL PopularitY of Programming Language index,” 2015. [Online]. Available: http://pypl.github.io/PYPL.html. [Accessed: 09-May-2016].

[37] T. Goldschmidt, R. Reussner, and J. Winzen, “A case study evaluation of maintainability and performance of persistency techniques,” in *Proceedings of the 13th international conference on Software engineering - ICSE ’08*, 2008, p. 401.

[38] V. Basili, G. Caldiera, and R. H Dieter, “Goal question metric (gqm) approach,” *Encycl. Softw. Eng.*, vol. 2, 1994.

[39] Z. Zhou and Z. Chen, “Performance Evaluation of Transparent Persistence Layer in Java Applications,” in *2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2010, pp. 21–26.

[40] M. C. Cuervo, D. I. P. Parra, and A. C. A. Aldana, “Evaluación y análisis de rendimiento de los frameworks de persistencia Hibernate y Eclipselink [Assessment and analysis of performance persistence frameworks Hibernate and EclipseLink],” *Ventana Informática*, no. 24. 21-Jun-2011.

[41] P. Van Zyl, D. G. Kourie, and A. Boake, “Comparing the performance of object databases and ORM tools,” in *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing couuntries - SAICSIT ’06*, 2006, pp. 1–11.

[42] K. E. Roopak, K. S. S. Rao, S. Ritesh, and S. Chickerur, “Performance Comparison of Relational Database with Object Database (DB4o),” in *2013 5th International Conference on Computational Intelligence and Communication Networks*, 2013, pp. 512–515.

[43] A. R. Fayyaz and M. Munir, “Performance Evaluation of PHP Frameworks (CakePHP and CodeIgniter) in relation to the Object-Relational Mapping, with respect to Load Testing,” Blekinge Institute of Technology, 2013.

[44] O. Leisalu, “Comparative Evaluation of Two PHP Persistence Frameworks,” 2009.

[45] V. Nagy, “Performance Analysis of Relational Databases, Object-Oriented Databases and ORM Frameworks,” University of Skovde, 2014.

[46] G. Booch, I. Jacobson, and J. Rumbaugh, “OMG Unified Modeling Lenguage Specification,” 2000. [Online]. Available: http://www.omg.org/spec/UML/. [Accessed: 20-Apr-2016].

[47] P. Wegrzynowicz, “Performance antipatterns of one to many association in hibernate,” *IEEE Trans. Softw. Eng.*, pp. 1475–1481, 2013.

[48] D. A. Alvarez and F. Arango, “Hibernate and spring - An analysis of maintainability against performance,” *Rev. Fac. Ing.*, vol. 0, 2016.

[49] L. Constantine and L. Lockwood, *Software for use: a practical guide to the models and methods of usage-centered design*. 1999.

[50] IEEE, “IEEE SA - 24765-2010 - Systems and software engineering -- Vocabulary.” [Online]. Available: http://standards.ieee.org/findstds/standard/24765-2010.html.

[51] E. Yourdon and L. Constantine, “Structured design: Fundamentals of a discipline of computer program and systems design,” 1979.

[52] I. Jacobson, “Object Oriented Software Engineering: A Use Case Driven Approach,” 1992.

[53] Hibernate, “Chapter 7. Collection mapping,” *Community Documentation*, 2015. [Online]. Available: https://docs.jboss.org/hibernate/orm/3.6/reference/en-US/html/collections.html. [Accessed: 18-Dec-2015].

[54] Sun, “Core J2EE Patterns - Data Access Object,” *Sun Microsystem*, 2002. [Online]. Available: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html. [Accessed: 25-Apr-2015].

[55] IEEE Spectrum, “The 2015 Top Ten Programming Languages,” 2015. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages. [Accessed: 11-Jun-2016].

[56] S. D. Network, “Java SE Desktop Technologies - Java Beans,” *Sun Microsystems*. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html. [Accessed: 25-May-2015].

[57] Grupo de Investigación en Ingeniería de Software, “Learning of Web Application Frameworks.” [Online]. Available: http://xue.medellin.unal.edu.co/grupois/wiki/index.php/Main_Page.