redin Revista Facultad de Ingeniería, Universidad de Antioquia, No. 80, pp. 97-108, 2016

Hibernate and spring - An analysis of maintainability against performance

Hibernate y spring – un análisis mantenibilidad contra desempeño

Danny Alejandro Alvarez-Eraso[1*], Fernando Arango-Isaza[1]

Facultad de Minas, Universidad Nacional de Colombia. Carrera 80 # 65-223 - Núcleo Robledo. A. A. 1027. Medellín, Colombia.

ARTICLE INFO

Received June 09, 2015
Accepted April 11, 2016

KEYWORDS

ORM, performance, spring framework, hibernate, maintainability

ORM, desempeño, spring framework, hibernate, mantenibilidad

ABSTRACT: Web application frameworks and ORM tools reduce time and effort needed when developing quality applications; but, since they are numerous and heterogeneous choosing the best suited is not an easy task. The comparative studies of these tools do not consider case studies of the necessary complexity to precisely measure their advantages and disadvantages. In order to contribute to the solution of this problem, we measured the HIBERNATE ORM response times for different queries in a rather complex case study with different database sizes, and compared the results with the ones obtained by using manually coded queries. Our comparison is relevant because even though ORMs are an important maintainability factor, not optimal queries can lead to bottle necks.

RESUMEN: Los frameworks para el desarrollo de aplicaciones web y las herramientas ORM permiten reducir el tiempo y esfuerzo al producir aplicaciones de software de calidad. Se han hecho estudios comparativos sobre estas herramientas pero dado que son numerosas y heterogéneas, escoger la más adecuada no es fácil. Hay estudios comparativos sobre estas herramientas, sin embargo no consideraron un dominio suficientemente complejo que permitan medir más precisamente sus ventajas y desventajas. Para aportar en la solución, comparamos el tiempo de respuesta de diferentes consultas en un dominio más complejo y con diferentes tamaños de base de datos. La comparación basada en este aspecto es importante ya que los ORM son un factor de mantenibilidad importante y porque consultas no optimizadas pueden conducir a cuellos de botella.

1. Introduction

Today, the task of producing web applications is widely addressed using a mixture of object oriented programming (OOP) [1], database (DB) products [2] and Web Application Frameworks (WAF) [3]. These techniques are used because they allow developers to represent data in a convenient way, increasing their productivity [1, 4-6].

However, objects and relational DBs belong to different paradigms, each one having its own data types and organization principles [7]. These differences have been called the impedance mismatch [7]. According to [8], Object Oriented Programming (OOP) languages are good at structuring complex non-persistent data having relationships, whilst relational DBs are good at structuring large amount of simple persistent data, but not good when dealing with complex associations.

To solve the impedance mismatch problem an application requires an interface layer to make both paradigms compatible. In WAFs, this layer has been called Object Relational Mapping (ORM). The ORMs map data from DB to OOP, allowing programmers to query DB records in a simple and powerful way without changing the OOP program point of view.

As mappings, ORMs can be used as abstraction barriers [9] between relational DBs and OOP programs. This is possible because, changes in the OOP or in the DB models affect the mapping function only, preventing them from propagating to the other side. Consequently, using ORMs is an important maintainability factor.

The maintainability benefits of ORMs have been included in most WAFs as a "off the shelf" pre-coded component. Unfortunately, off the shelf ORMs might threaten the overall application performance. In fact, when a programmer builds his own ORM, he is free to tailor it to the program, applying specific efficiency criteria to improve the query performance; in contrast, off the shelf ORMs craft the queries in a standardized way, making harder to achieve the same specific efficiency.

Consequently, using off the shelve ORMs creates tension between maintainability and performance. On one hand, they ease the task of developing websites controlling the complexity of the software specification; on the other hand,

* Corresponding author: Danny Alejandro Alvarez Eraso
e-mail: daalvareze@unal.edu.co
ISSN 0120-6230
e-ISSN 2422-2844

DOI: 10.17533/udea.redin.n80a11

they can affect performance to the point that the program response times become unacceptable.

That tension has been studied by several authors [10-13] for the Hibernate ORM Framework [14]. However, they did not consider a case study complex enough to properly analyze the tension we described before.

In our study, we went one step forward by analyzing the performance for queries of three complexity levels and three different DB sizes. We compared queries manually coded in SQL, optimized and non-optimized (in what follows optimized and non-optimized SQL queries) against queries automatically crafted by Hibernate (in what follows Hibernate queries). To test the queries we built a program in the Spring WAF. We selected Spring based on its popularity [15-18] and because it integrates easily with Hibernate.

As the main goal of our study was to analyze the impact of query-complexity and DB size in the performance of Hibernate and SQL queries, we took measures to maintain constant all other variables. In particular, we used default configurations for Hibernate, Apache and MySQL, avoiding tune-ups. Moreover, to be sure we followed the best possible coding practices, we consulted the official documentation, online forums, taking the recommendations from [19] regarding Hibernate annotations.

We present our research as follows: section 2 analyzes the related work; section 3 presents our case study problem; section 4 presents the problem Class Diagram and relational model; section 5 shows our efficiency findings. In section 6 and 7 the reader can find the details about the implementation: section 6 describes the Spring JAVA Classes and the way they relate to the Relational DB Tables, and section 7 describes the DB access elements built as JAVA DAOs. Finally, in section 8 we present conclusions and future work.

2. Related work

In [10], 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 DB sizes or optimized SQL queries.

The authors in [11] highlight the role of the "Persistent Framework" (the ORM) as a layer between a DB and the business classes (see Figure 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.

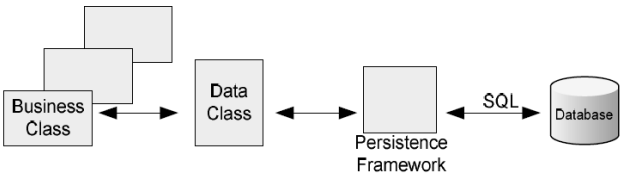

Figure 1 Persistence framework mode - from [11]

Similarly, in [12] 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 this setting.

In [13], 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 DB 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.

In [17], the authors tested db4o and SQL by reading and writing on DBs of different sizes. They concluded that for low DB sizes db4o performance was closer to MySQL, whilst Hibernate was significantly slower. However, when DB 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, and scale closer to MySQL.

The studies presented in [10-13] have the inconvenience that they focused on a single class/table and do not explore the complexities related to relationships. Those studies also present little implementation details, making difficult to reproduce the experiments.

Even when the study presented in [17] considered relationships, they are reduced to relationships between two class/tables (flights and airports), and the DB sizes were not large enough. So, even when Hibernate has a satisfying behavior, the evidence is not sufficient to say that it will remain as scalable as SQL queries.

3. Case study definition

3.1. Class diagram

In Figure 2 we present the Class Diagram of an advertisement web application oriented to display photo albums. We drew the diagram using the notation proposed in [20].

Note that our CD has classes and class member names written in Spanish. This is because the source code names were written in that language and we did not want to misrepresent what was done before.

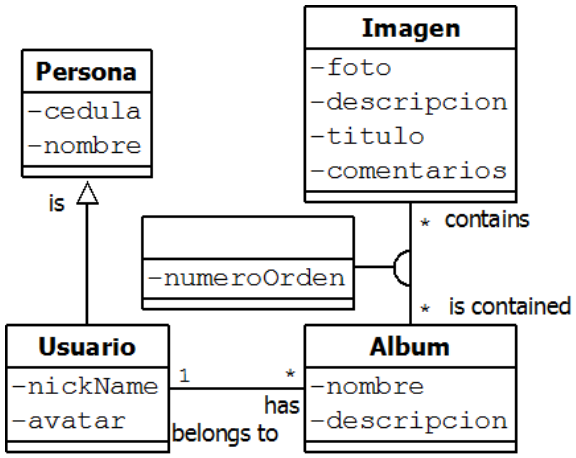

Figure 2 Class Diagram

3.2. Complexity levels of queries

In our study, we related the query complexity level, to complexity of the result obtained by reading the DB and turning the retrieved data into objects with relationships. Based on this line of thinking, we used three query complexity levels, each one associated to the effort needed to obtain the object structure represented in Figure 3.

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

Level 1 - Persona

Level 1 is our lowest query complexity level. In this level, we retrieve a list of *Persona* objects. We consider low this query complexity level, 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.

Level 2 - Usuario and album

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

Level 3 - Album and imagen

Level 3 is our highest query complexity level. In this level, we retrieve a set of *Usuario*, *Persona*, *Album* and *Imagen* objects, as well as the image order number (*numero Orden*, see Figure 2). This query level occurs, for example, when we want to know what images are in albums of some user.

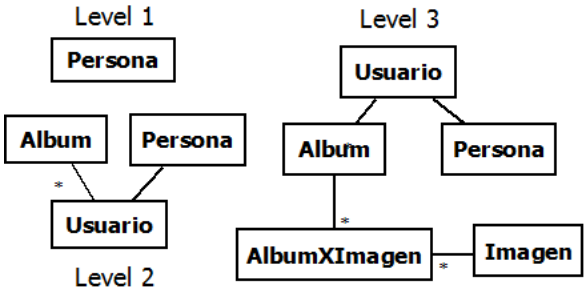

Figure 3 Goal object tree for each query level

This is the most complex scenario for two reasons. In the first place, increasing the search depth in a DB implies accessing more tables, and this is a process that can take a long time if the DB 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. Designing the case study

We built a Spring web application following the Model-View-Controller (MVC) design pattern [21].

4.1. Database structure

In concordance to our CD, our DB has five tables, one for each class in our CD and one intersection table to represent the many-to-many relationship. In this way, the relationship attribute *numeroOrden* was represented with an extra column in the intersection table. In Figure 4 we show our DB relational model.

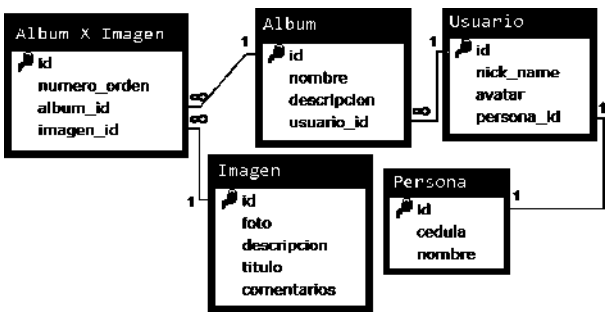

Figure 4 Relational model for our case study

4.2. Class model in spring framework

Given that Hibernate does not allow linking classes to multiple tables, we decided to code a JAVA class for every DB table. Consequently, we defined {*Album*, *Imagen*, *Usuario* and *Persona*} classes. In order to represent the primitive data from the DB tables, we used private *String* attributes in the classes. Finally, we included a *Long id* attribute as the Hibernate identification strategy.

4.3. Relationships of the class model

In the Spring code, we represented relationships between two classes with object valued attributes of the corresponding type in both classes [22]. The relationship's cardinality determines if the attribute is a single object or a collection of objects.

For one-to-one relationships, we included an attribute of the other side 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 type. For example, the one-to-many relationship between *Usuario* and *Album* was represented with one *Usuario* typed attribute in *Album* and one *List<Album>* typed 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 DB with an intersection table containing an extra column, we decided to use a new class (*AlbumXImagen.java*) containing an attribute for *numeroOrden*. Additionally, to maintain consistency, from the point of view of *Album* and *Imagen* classes, we included an *AlbumXImagen*-typed attribute in both of them, as shown in Figure 5.

```
// Joining relationship
  private List<AlbumXImagen>
       albumXimagenList; // ...
```

Figure 5 Joining attribute placed in *Album* and *Imagen* Classes

We represented the inheritance relationship between *Usuario* and *Persona* using object valued attributes instead of using JAVA inheritance. We did so because, no matter if 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. Results

The following results show the impact of the query complexity in the performance of Hibernate and both optimized and non-optimized SQL queries. They were obtained by running our application for the three complexity and three different DB sizes.

To run the application we used Apache 2.4.4, MySQL 5.6.12 and Hibernate 4.2.2. Final software versions. Also, the experiments were made in a Samsung ultrabook, Intel Core I5-3337U CPU @1.80GHz x 4, 4GB DDR3 RAM, and 64 bits Windows 8. We ran all test the same day, in the same machine, with the same processes load, avoiding tune-ups that could give advantage to Hibernate or SQL queries.

We performed reading tests for each complexity level defined in section 3.2. Tests were executed in a DB with 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 for each successive repetition of the test showing a tendency to descend. Those changes occur as a consequence of the cache mechanisms that are present in Spring, Hibernate and MySQL. Even the operative 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 memory.

Consequently, we present our results in two steps. The first step presents the experiment raw data using time vs iteration graphs. The second step compares performance relying on the average response time. 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. The actual results for every reading level and DB size are presented below.

5.1. Results per complexity level

Level 1 results for 100,000 tuples are shown in Figure 5, for 500,000 in Figure 6 and for 1,000,000 in Figure 7.

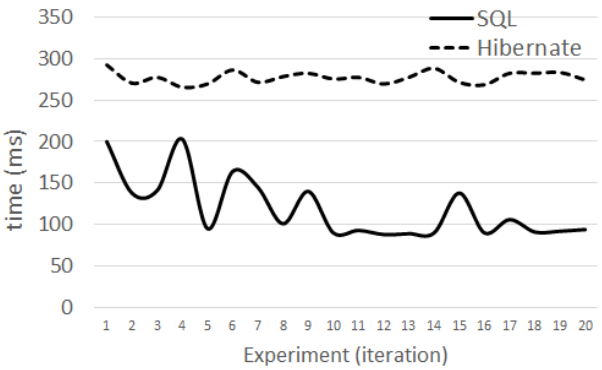

Figure 6 Level 1 queries with 100,000 tuples

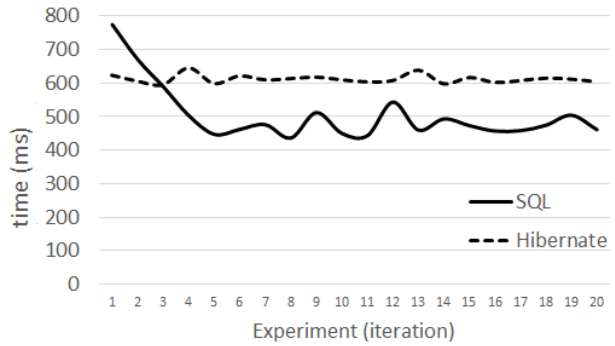

Figure 7 Level 1 queries with 500,000 tuples

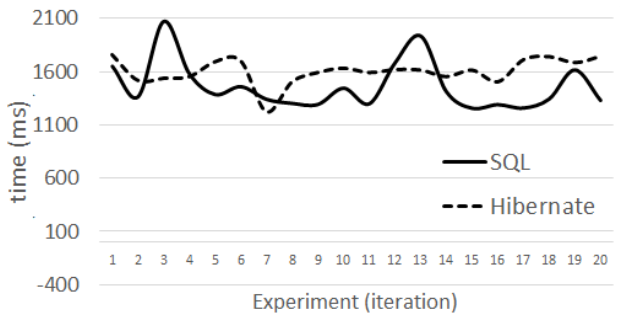

Figure 8 Level 1 queries with 1,000,000 tuples

Level 2 results for 100,000 tuples are shown in Figure 8, for 500,000 in Figure 9, and for 1,000,000 in Figure 10.

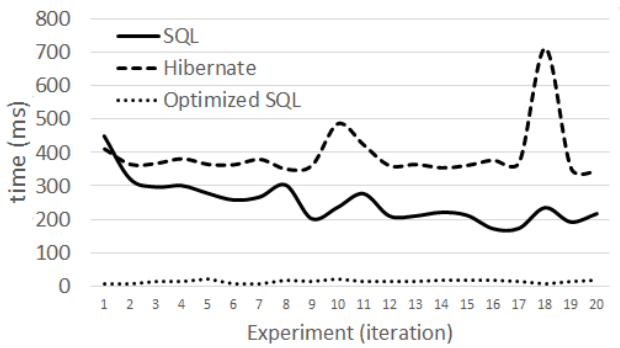

Figure 9 Level 2 queries with 100,000 tuples

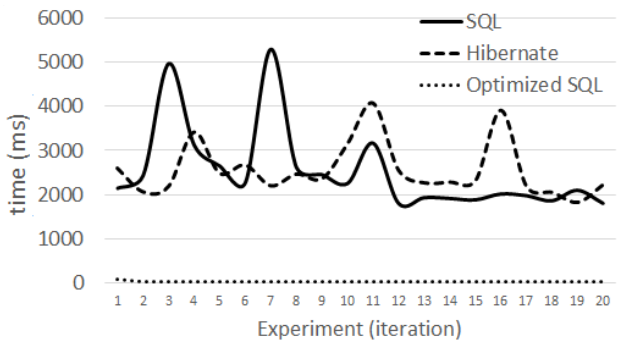

Figure 10 Level 2 queries with 500,000 tuples

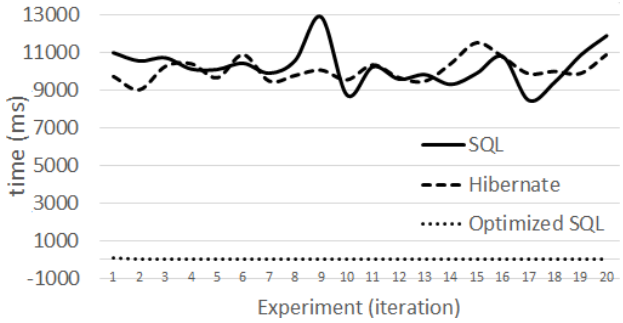

Figure 11 Level 2 queries with 1,000,000 tuples

Level 3 results for 100,000 tuples are shown in Figure 11, for 500,000 in Figure 12 and for 1,000,000 tuples in Figure 13.

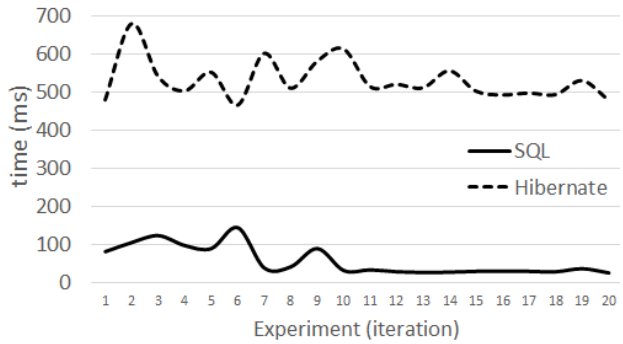

Figure 12 Level 3 queries with 100,000 tuples

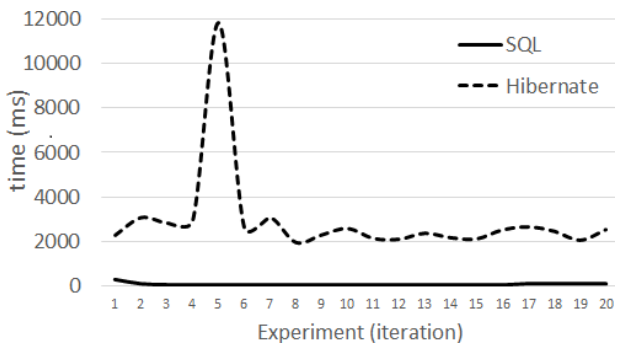

Figure 13 Level 3 queries with 500,000 tuples

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

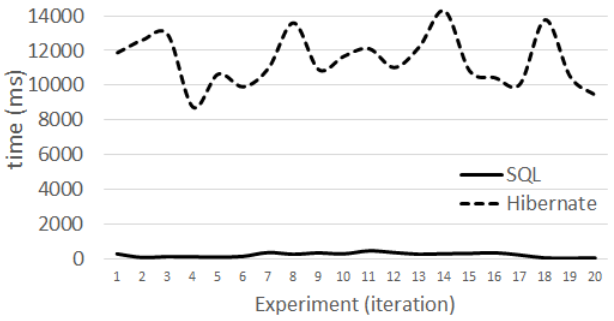

Figure 14 Level 3 queries with 1,000,000 tuples

Table 1 Average 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

5.2. Discussion

As the DB size grows, the application's response time grows too. This occurs because the search space gets bigger and the *Join* clauses are especially sensitive to the amount of records. In Figure 14, 15 and 16 we present the average response time in function of the database size.

Our average response times, for levels one and two, show that the performance of Hibernate and non-optimized SQL queries is similar in a DB with few records and simple relationships. This behavior was already described by [17]. However, when going from half million records to one million, we can see that Hibernate and non-optimized SQL queries' response time is strongly non-linear. This non linearity is shown for level 1, 2 and 3 in Figure 14 and 15, where performance times grow tree and almost five times respectively.

In fact, in Level 1 with DB sizes of 100,000, 500,000 and 1,000,000 tuples, Hibernate is in average 158 (132%), 107 (21%), and 140 (10%) ms slower respectively. In Level 2 with the same DB sizes Hibernate in average is 141 (56%) and 44(2%) ms slower than the not optimized SQL respectively. Our findings for 1,000,000 seem odd as Hibernate was faster for 191 ms (-2% faster). This was probably caused by the cache capabilities of all of the technologies we used, and because in this test level the SQL query is not optimized. Furthermore, the reader must remember that all of our tests include tuple selection based on a randomly generated name.

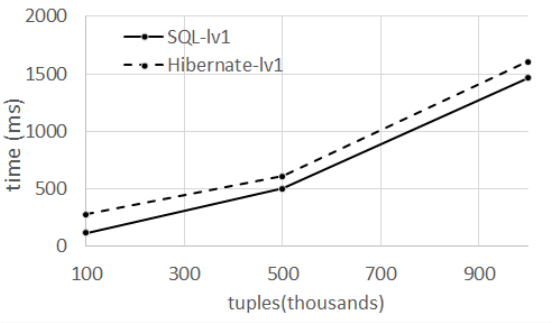

Figure 15 Level 1 average time in function of the DB size

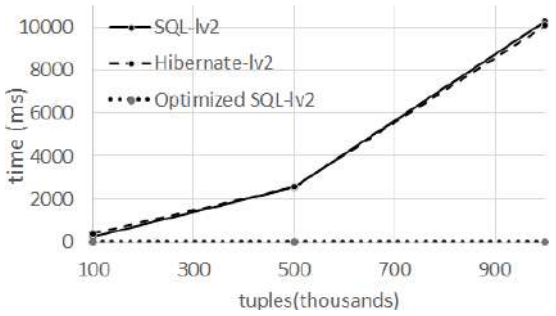

Figure 16 level 2 average time in function of the DB size

On the other hand, authors in [17] 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 15 and 16). In Level 2, when we optimized the SQL query with DB sizes of 100,000, 500,000 and 1,000,000 tuples, Hibernate is in average 378.8 (2,859%), 2,555.8 (13,853%) and 10,066.35 (58,525%) ms slower respectively. In Level 3, with the same DB sizes Hibernate, is in average 475 (842%), 2.855 (5,079%) and 11,201 (5,139%) ms slower respectively. There is a big difference in this case.

In contrast as we did with SQL, we could not identify a good way of optimizing level 3 queries using Hibernate. We could not even find specific HQL´s documentation regarding this topic. So, to the best of our knowledge, we made sure that Hibernate Level 3 queries minimized unnecessary DB accesses.

However, another unexpected finding was that hibernate reports doing 26 queries to obtain the results for our level-three-complexity scenario. In Figure 17 we show a fragment of the Spring console output of those undesirable queries.

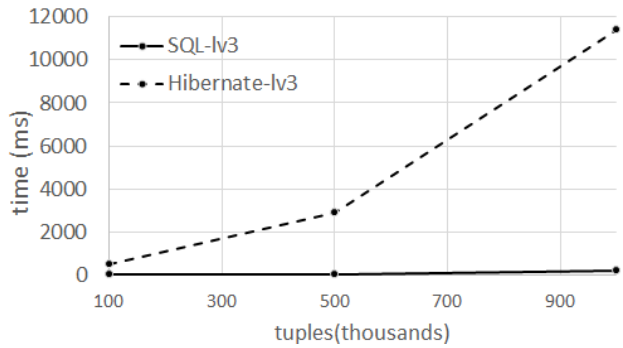

Figure 17 Level 3 average time in function of the DB size

```
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_
```

Figure 18 Hibernate console output for level 3 scenario

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

Figure 19 Linking a class and table - Album case

Finally, as the details behind our results can be of particular interest to certain readers we present them as follows, in section 6 we show how we linked Hibernate to our CD and in section 7 we present how the DB was accessed with both Hibernate and SQL.

6. Implementation

We made sure the most convenient implementation for our tests were achieved by focusing on query complexity and DB size be the only variables impacting performance. So, we searched all the available literature we could and followed the official documentation, online forums and the recommendations of [19] regarding aspects influencing the performance of Hibernate. The authors in [19] define an anti-pattern as a bad practice that has a negative impact in application performance, and presents five of them regarding relationships. On this ground, we constructed the best possible implementation based on the acquired knowledge.

6.1. Linking classes and tables

Hibernate can establish a link between a JAVA class and a DB table by using annotations. Annotations in the class definitions link them to the corresponding DB 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 DB tables using *@Entity, @Table, @Id, @GeneratedValue* for Hibernate identification strategy, and *@Column* for the primitive class members. In Figure 19 we present Album's case as an example.

For one-to-one and one-to-many relationships we used *@ oneToOne, @OneToMany and @ManyToOne annotations*. In all cases, we defined the inverse relationship using the @ *JoinColumn* annotation.

6.2. Many to many relationships

To link this relationship, Hibernate offers *@ManyToMany* annotations, but in our class diagram this relationship has an attribute. Because of this, we split such relationship in two one-to-many relationships.

This caused *Album* and *Imagen* classes to be linked through *AlbumXImagen*. So, from *AlbumXImagen* class' point of view this relationship becomes two new ones whose cardinality resembles an intersection table. This is, we put two *@ manyToOne* and two *@JoinColumns* annotation in this linking class.

7. Database access

7.1. Controllers and DAOs

We used controllers with two main purposes. In first place, to execute DB queries either using SQL or Hibernate; and in second place, to measure and report response times by using *Date.getTime()* before and after of the connection, querying, and object assembly processes. These controllers serve the HTTP request using a Data Access Object (DAO) [23]. We built two DAO classes one using Hibernate and another using SQL.

Our DAO classes access the DB using *@autowired* BEANs [24]. Hibernate DAO uses the *sessionFactory* BEAN and *@ Repository* annotation to manage DB transaction whilst SQL DAO uses the MySQL *dataSource* BEAN.

```
// 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 obj tree
```

Figure 20 Hibernate DAO structure in our implementation

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

Figure 21 Level 1 criteria definition

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

Figure 22 Level 2 criteria definition

7.2. Object trees with 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 20. This *criteria* was 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.

level 1 Hibernate reading

In this level, we retrieve a List of *Persona*. So, we defined a *Criteria* with *<OBJ_CLASS>* equals to *Persona* (see Figure 21), and we added the name and page size restrictions (see section 3.2). Here, the variable *nombre* contains a random char between [a-z] (same for level 2 and level 3).

Level 2 Hibernate queries

In this level, we retrieve a List of *Usuario* including relationships with *Persona* and *Album*. So, we defined a *Criteria* with *<OBJ_CLASS>* equals to *Usuario* (see Figure 22) and, in addition to the name and page size restrictions, we added relationship constrains (see Figure 4) making sure I*magen or AlbumXImagen* data was not retrieved.

Level 3 Hibernate queries

In 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 scenario reads data from all tables in the DB and construct the corresponding object tree.

Our first attempt to build the required *criteria* was to set *<OBJ_CLASS>* equals to *Usuario* and, in addition to level 2 query restrictions, we modified *AlbumXImagen's* fetch mode from *SELECT* to *JOIN*. However, this approach resulted in this execution exception: *"Request processing failed; nested exception is org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple bags"*. Not being clear from the Hibernate's documentation the nature of this error, we assumed it was caused by the join depth complexity. The reader must remember that every *Usuario* has a List of *Albums* and every one of those also has a List of *AlbumXImagen*, see Figure 3.

After digging a little more in the problem we solved the issue by adjusting our model's classes. This is, we changed both *@OneToMany* and *@ManyToOne* annotations so the related data would be eagerly loaded. Thus, we modified the *albumXimagenList* attribute in *Album and imagen* classes as a non-lazy collection, in Figure 23 we show this modification in the Java file.

This approach has the inconvenient that even when it solves the execution problem, the way classes were mapped causes both *album* and *albumXimagenList* attributes are always loaded. This lead queries to poor performance when those data are not required, take our level 2 queries for example. On this ground, we defined the required *criteria* as shown in Figure 24:

```
// Album.JAVA file
@OneToMany(mappedBy = "album")
@LazyCollection(LazyCollectionOption.FALSE)
private List<AlbumXImagen>albumXimagenList;
```

Figure 23 Setting the joining attribute as non-lazy Collection

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

Figure 24 Level 3 criteria definition

```
SELECT * from persona
 WHERE nombre like '%nombre%'
 ORDER BY nombre DESC LIMIT 0, 50"
```

Figure 25 Level 1 SQL query

7.3. Object trees with SQL

All data obtained with manually coded SQL queries will be used as a reference to assess Hibernate´s results. To achieve the best performance in this DAO, we always accessed the DB using a single query, and building the required object tree with that result only.

Additionally, all of the three DAO methods are structured in a similar way; the two code sections that change correspond to the executed SQL statement and the construction of the object tree.

Object assemble using typed maps

To be able to compare SQL response times with those from Hibernate, it is necessary to recreate the same conditions. Given that *executeQuery()* method responds with a *ResultSet* collection instead with a set objects, we need to transform that *resultSet*.

Transforming tuples into objects is a simple task when the resulting data belongs to a single class (like in level 1 scenario); however, the same task becomes harder when such data belongs to many classes (like in level 2 and 3 scenarios).

To reduce the time the program takes to transform level 2 and 3 *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. This approach solves the problem with and algorithm with order O(n).

Level 1 SQL queries

In this level, we retrieve a List of *Persona using* the SQL query shown in Figure 25. Also, we assembled the object tree from this *Resultset* using a simple loop that creates a *Persona* instance containing every row field as attributes.

Level 2 SQL queries

In this level, we retrieve a List of *Usuario* with *Persona* and *Album* data using the SQL query shown in Figure 26. Also, we assembled the object tree from this *Resultset* using one *Usuario* typed map.

Although this code is totally functional, the query can be optimized. We did this optimization as an attempt to prove that Hibernate cannot find the best way of solving the join order execution. We present our optimized SQL query in Figure 27.

Level 3 SQL queries

In this level, we retrieve a List of *Usuarios* with *Persona*, *Album*, *Imagen* and *AlbumXImagen* data using the SQL query shown in Figure 28. Also, we assembled the object tree using two typed maps, one for *Usuario* and one for *Album*.

```
SELECT p.id AS persona_id,
 p.nombre AS persona_nombre,... FROM usuario
 u LEFT JOIN persona p ON p.id = u.persona_id
 RIGHT JOIN album al ON al.usuario_id = u.id
 WHERE p.nombre like '%nombre%'
 ORDER BY p.nombre DESC LIMIT 0, 50"
```

Figure 26 Level 2 SQL query

```
SELECT uxp.persona_id, uxp.person_nombre,... FROM((SELECT u.id, p.id, ...
 FROM usuario u LEFT JOIN persona p
 ON p.id = u.persona_id WHERE
 p.nombre like '%nombre%' LIMIT 0, 50 ) uxp)
LEFT JOIN album al ON al.usuario_id = uxp.id
ORDER BY uxp.persona_nombre DESC
```

Figure 27 Level 2 Optimized SQL query

```
SELECT uxp.persona_id, uxp.persona_nombre...
FROM (( SELECT u.id, u.avatar, ...
 FROM usuario u LEFT JOIN persona p
 ON p.id = u.persona_id WHERE
 p.nombre like '%nombre%' 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.persona_nombre DESC
```

Figure 28 Level 3 SQL query

8. Conclusions

In this research we compared the performance of Hibernate and SQL queries. 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 Hibernate and SQL queries maintaining the same conditions for the three query complexity levels and DB sizes. To that end, we ran all tests the same day, in the same machine, with the same process load, using default configurations for Spring, Hibernate, Apache and MySQL, and maintaining the same DB 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 Hibernate 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 [17]). On the contrary, when the query complexity grows Hibernate proved to be much slower than optimized SQL queries. This happens because Hibernate cannot find the optimal join execution order, when accessing multiple tables with a relationship-deep of two levels or more.

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. 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 DB tables and ease the task of querying them with a single sentence.

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 DB products as it exhibits the n+1 queries problem.

Secondarily, even though we performed tests for three DB sizes only, Hibernate and non-optimized SQL queries' average response time vs DB-size curves appear to be strongly non-linear. Plus, the ratio of optimized-SQL and Hibernate-average-response-time is not constant and grows with the DB size (see Figure 15 and 16), suggesting that they belong to different complexity orders. Clearly there is a big gap between both Hibernate and non-optimized SQL queries, when compared to optimized SQL queries. However, more research is needed to properly determine the complexity order.

Finally, we want to point out that we had to change our class model to be able to retrieve data in level 3 scenario when using Hibernate. Even when this change was needed, it sacrifices the semantics of other queries. This is, by making eager the load of two attributes in level three complexity level, we also force level two and level one to load those attributes when they are not required. This is an issue we expect to solve in the future.

As future work, we will study more in depth the causes that lead Hibernate to poor performance. Additionally, we will test other ORM tools for JAVA and from other languages to determine if this problem is specific to Hibernate or transversal. To do so, we will use our case study as a benchmark.

9. References

1. G. Booch, "Object-oriented development", *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, pp. 211–221, 1986.
2. V. Srinivasan and D. Chang, "Object persistence in object-oriented applications", *IBM Syst. J.*, vol. 36, no. 1, pp. 66-87, 1997.
3. I. Vuksanovic and B. Sudarevic, "Use of Web Application Frameworks in the Development of Small Applications", in *34th Int. Conv. MIPRO*, Opatija, Croatia, 2011, pp. 458–462.
4. G. Low and S. Huan, "Impact of object oriented development on software quality", in *9th International Workshop Software Technology and Engineering Practice*

(STEP), Pittsburgh, USA, 1999, pp. 3–11.
5. N. Wilde and R. Huitt, "Maintenance support for object-oriented programs", *IEEE Trans. Softw. Eng.*, vol. 18, no. 12, pp. 1038–1044, 1992.
6. 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, pp. 77-85, 2002.
7. C. Ireland, D. Bowers, M. Newton and K. Waugh, "A Classification of Object-Relational Impedance Mismatch", in *1st International Confernce on Advances in Databases, Knowledge, and Data Applications* (DBKDA), Gosier, France, 2009, pp. 36–43.
8. C. Murdaca, "An Object-Relational Compiler", in *2009 WRI World Congress on Computer Science and Information Engineering*, Los Angeles, USA, 2009, pp. 438–442.
9. H. Abelson, G. Sussman and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, USA: The MIT Press, 1996.
10. M. Cuervo, D. Peñalosa and A. Alarcón, "Evaluación y análisis de rendimiento de los frameworks de persistencia Hibernate y Eclipselink", *Ventana Informática*, no. 24, pp. 9-23, 2011.
11. Z. Zhou and Z. Chen, "Performance Evaluation of Transparent Persistence Layer in Java Applications", *in International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery* (CyberC), Huangshan, China, 2010, pp. 21–26.
12. P. van Zyl, D. Kourie and A. Boake, "Comparing the performance of object databases and ORM tools", in *Annual research conference of the South African institute of computer scientists and information technologists on IT research in developing couuntries* (SAICSIT), New York, USA, 2006, pp. 1–11.
13. K. Roopak, K. Swati, S. Ritesh and S. Chickerur, "Performance Comparison of Relational Database with Object Database (DB4o)", in *5th International Conference on Computational Intelligence and Communication Networks* (CICN), Mathura, India, 2013, pp. 512–515.
14. Hibernate, *Hibernate. Everything data*. [Online]. Available: http://hibernate.org/. Accessed on: Mar. 10, 2016.
15. Hotframeworks, *Web framework rankings | HotFrameworks*. [Online]. Available: http://hotframeworks.com/. Accessed on: May. 26, 2015.
16. T. Shan, W. Bank and W. Hua, "Taxonomy of Java Web Application Frameworks", *IEEE International Conference on e-Business Engineering* (ICEBE), Shanghai, China, 2006, pp. 378-385.
17. V. Nagy, "Performance Analysis of Relational Databases, Object-Oriented Databases and ORM Frameworks", Bachelor Degree Project, University of Skövde, Skövde, Sweden, 2014.
18. J. Arthur and S. Azadegan, "Spring Framework for Rapid Open Source J2EE Web Application Development: A Case Study", in *6th Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and 1st ACIS International Workshop on Self-Assembling Wireless Network* (SNPD/SAWN), Towson, USA, 2005, pp. 90–95.
19. P. Węgrzynowicz, "Performance antipatterns of one to many association in hibernate", in *Federated Conference on Computer Science and Information Systems* (FedCSIS), Kraków, Poland, 2013, pp. 1475–1481.
20. Object Management Group (OMG), *Unified Modeling Language ™* (UML®). [Online]. Available: http://www.omg.org/spec/UML/. Accessed on: Apr. 20, 2016.
21. G. Krasner and S. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system", *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, 1988.
22. G. King, "Chapter 7. Collection mapping", in *HIBERNATE - Relational Persistence for Idiomatic Java,* 2004. [Online]. Available: https://docs.jboss.org/hibernate/orm/3.6/reference/en-US/html/collections.html. Accessed on: Dec. 18. 2015.
23. Oracle Corporation, *C*ore J2EE Patterns - *Data Access Object*, 2002. [Online]. Available: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html. Accessed on: Apr. 25, 2015.
24. Oracle Corporation, *Java SE Desktop Technologies*. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html. Accessed on: May 25, 2015.