MASTER'S THESIS | LUND UNIVERSITY 2018

Migration & Evaluation of Automatic Query Hint Generation Method in Persistent Systems

Erik Jonasson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-44

Migration & Evaluation of Automatic Query Hint Generation Method in Persistent Systems

Erik Jonasson
dat12ejo@student.lu.se

December 6, 2018

Master's thesis work carried out at itestra GmbH.

Supervisors: Per Andersson, per.andersson@cs.lth.se
Arnaud Fietzke, fietzke@itestra.de

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Object-relational-mapping tools are widely used in software development to translate between object oriented programming languages and relational databases. They serve as an abstraction layer between the business logic and the underlying data layer. However, if these tools are misconfigured in terms of *fetch strategy*, they can cause performance losses in the shape of the $N + 1$ problem, which means that every association of an object gets loaded with a separate query. In systems where network latency exist, more executed queries will negatively impact performance. In this report a method to automate the configuration of fetch strategies has been evaluated. The obtained results from this study shows that the method can simplify the configuration for the developer. Moreover, the automated configuration ended up eliminating 11.3% of the queries, theoretically improving performance in systems with network delay. With the simplicity of use, the tool offers increased performance in terms of total execution time and amount of executed queries, without spending the time that manual tuning would require. However, the manually configured version performs better compared to the version with automated prefetching. In its current state, the tool would benefit from continued development and official support from the Hibernate team.

Keywords: Persistence, Relational database, Fetch strategies, Lazy loading, Autofetch

Acknowledgements

I wish to thank itestra GmbH and specifically Arnaud Fietzke for hosting my master thesis and for all the help that has been provided to me. Special thanks is also carried out to Per Andersson and Flavius Gruian for supervising and examinating this thesis respectively. Secondly I would like to thank Vlad Mihalcea for assisting with his expertise in the Hibernate platform, and to Adriano Machado for all the assistance regarding the implementation and migration of the tool.

Contents

Chapter 1
Introduction

1.1 Background

Companies today use IT infrastructures that are often large and complex with the capacity handling enormous amounts of data. Commonly, these systems use a so called *Distributed Client-Server Architecture*, implying that the different system components are separated in different locations. These systems usually rely on *object persistence*, meaning that the objects created within the application have a lifespan longer than the execution of the program and are stored in some way using some type of database, enabling save and restore data operations in the system. *Orthogonal Persistence* takes the concept one step further, implying that the retrieval and save operations are automatic within the application without user interaction[1, 2].

With extensive processing of data in these type of systems, it is crucial that the processes run as efficiently as possible. However, the task of optimizing the data flow is challenging and can be approached from different angles. Historically, most of the tuning regarding this topic has been done through various optimizations of queries. These types of manual query optimizations are still an efficient way to tune the performance of a system, but tend to be error prone[3].

In large scale client-server systems, one of the more expensive operations is to fetch data from the databases. Depending on what type of database that is used, the level of complexity of the task can vary. In the case of object-relational and object databases, the data is already stored as objects and can therefore be fetched easily. However, in the case of relational databases, the challenges are bigger since the data is simply stored as sets of data. The difference in the underlying paradigms of these two systems is often called the *object-relational impedance mismatch*, a complex phenomenon consisting of numerous problem aspects that does not have any clear solutions due to the problem's many-sided nature[5, 6]. In Ireland & Bowers (2015) they present some arguments why the impedance mismatch is a *wicked problem*, meaning that the problem does not have a definitive formulation and

proposed solutions to the problems end up creating other problems.

There are methods to work around this however, the most common one being *Object-Relational-mapping* software, from now on referred to as *ORM-software*. This software translates between the object oriented model and the relational model, making the transition between these two models transparent, normally called *transparent persistence*.

The performance concern in these systems is not only related to what type of database that should be used, but also how data should be stored and how to configure the system correctly based on the characteristics of said system. A configuration adapted to the specific environment of the system could lead to noticeable performance improvements[7]. An operation that is usually subject to optimization in these types of systems is making fetching from the database more efficient, namely by adjusting *when* and *how* something should be fetched. This type of optimization requires knowledge and is time-consuming[9].

Depending on the scenario, factors such as network delay and fetching patterns can have a significant impact on system performance[10]. A common pitfall for bigger systems that will cause considerable performance drops, is the execution of numerous round trips in the database with single entry select queries. This problem is generally known as the *N + 1 problem* due to the amount of generated queries when iterating over an uninitialized collection in ORM software. The *N + 1* problem is a phenomenon which the evaluated method of this report strives to minimize. The cause of this problem is that for each loaded association of an entity from the database, a select query will be executed[11]. In systems with noticeable latency, this will cause lower performance than a method that would merge all these queries into a *batch* of queries or simply one query containing joins between associations. The situation that occurs in a system with network latency can be seen in Figure 1.1, where a query is being sent over a connection, is being executed by the database, and then finally the results being returned over the connection. Potential network delay will affect performance for each query being sent over the network. The higher the network delay, the worse the performance. Since this will affect all executed queries, we want to minimize the number of executed queries as much as possible.

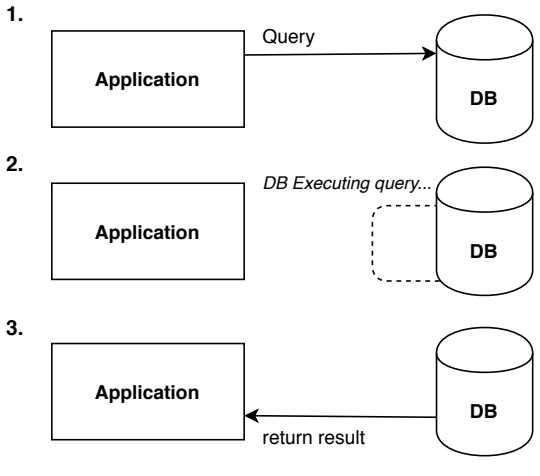

Figure 1.1: Illustration of how each executed query will affect performance of a system

1.1.1 ORM-tools

In the Software development industry, the most common type of database is the relational database[15]. With theoretical foundation in set theory and with its long time on market, it has become a common option for many big enterprise systems. As a result of the relational database's popularity, the object-relational impedance mismatch is a present concern for most developers. To tackle the problem, it is common to pair the relational database with an ORM-tool, such as *EclipseLink* and *Hibernate*[16, 17]. In the case of Java applications, which are the focus of this report, the ORM-concept has been formalized in the form of the *JPA-interface*, abbreviation for *Java Persistence API*, which strives to unify all Java based ORM-tools under one API[18].

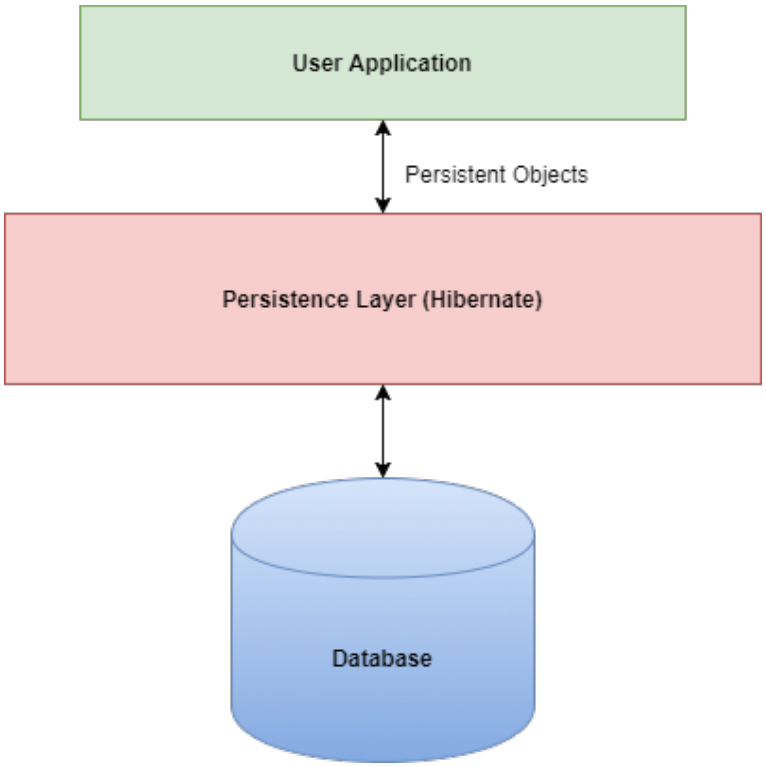

Figure 1.2: Basic ORM architecture

A commonly mentioned benefit of ORM-tools is that the extra data layer makes the data insertion and retrieval more portable and abstract, completely separating data access logic from business logic in applications[24]. The extra layer added by ORM-software can be observed in Figure 1.2. This extra layer implies that the developers do not have to think about writing data access logic in the business logic. All those operations are carried out in the background when the user for example loads an object. Compared to the other commonly used way of interacting with the database level in Java, *JDBC* for example, it requires less code to accomplish the same operation[30]. In JDBC, the user has to manually handle connection handling and caching among other things.

Furthermore, since developers do not have to write queries themselves the system can be easier to maintain[24]. In figure 1.3, sample code from Hibernate is presented, showing the simplicity of persistence in ORMs compared to JDBC seen in Figure 1.4.

```
sess = openSession();
tx = sess.beginTransaction();
Employee root = new Employee( "Arnaud", null, null, new Address( "Teststrasse 1",
↪  "Munich", "Germany" ));
Employee e0 = new Employee( "Erik", root, null, new Address( "Backvagen 21",
↪  "Boras", "Sweden" ));
root.addSubordinate(e0);
sess.save(root);
tx.commit();
sess.close();
```

Figure 1.3: ORM sample code, saving an object to the database with an association

```
try {
    conn = DriverManager.getConnection(DB_URL, USER, PASS);
    stmt = conn.createStatement();
    String sql = "
    insert into Employee (Munich, Germany, Teststrasse 1, null, Arnaud, null, 1);
    insert into Employee (Boras, Sweden, Backvagen 21, null, Erik, 1, 2);
    update Employee set supervisor_id = 1 where employee_id = 2";
    ResultSet rs = stmt.executeQuery(sql);
    rs.close();
} catch(SQLException se){
    se.printStackTrace();
}
```

Figure 1.4: JDBC sample code, saving an object with an association

There are disadvantages however. Firstly, due to the abstraction, ORM-tools can potentially make developers less aware of the underlying database model and therefore what queries that are being carried out. Secondly, with ORM-software there will be extra overhead and therefore it might not fit all types of applications. Additionally, without knowledge and prior experience, these types of software can be difficult to set up properly to achieve desired performance, since the default behaviour of the different platforms may not be optimal for one specific use case[7]. Furthermore, maintaining ORM-code can be difficult due to the tendency of the API:s changing frequently, ultimately leading to maintainability issues. The lack of return type checking at compilation time causes the errors to be difficult to spot. Lastly, static code analysers are often not developed to detect faults in ORM-code, adding to the problems with using an ORM[8].

1.1.1.1 Hibernate

As previously mentioned, today there are numerous ORM-tools on the market, each with its different edge and features. ORM-tools are widely spread in most object oriented programming languages such as *C++*, *.NET*, *C#* and *Java*, with examples being *ODB*, *NHibernate* and *Ebean* respectively[12, 13, 14]. In the Java scene, the two most prominent are EclipseLink and Hibernate. In this study we have chosen to focus on Hibernate.

Hibernate is an open-source ORM-tool developed since 2001 containing a complete framework for persisting so called *POJOs*, *Plain Old Java Objects*. Similarly to other

ORM-tools, it maps object oriented domain model to the relational model used in relational databases. One of its main features is that it includes its own native API, meanwhile being an implementation of the standardized JPA specification, allowing for high flexibility in regard to portability and maintenance. Hibernate can be used with both JPA annotations and *xml*-files to map classes to tables in SQL. Furthermore, Hibernate uses its own type of query language called *Hibernate Query Language*, enabling users to query hibernate data objects in a SQL-like fashion[17]. Being an implementation of JPA 2.1 since version 4.3, it uses default settings according to the JPA-specification, such as prefetch directives for mappings. Hibernate and its extensions mostly use the GNU Lesser General Public License 2.1 license[19].

1.1.1.2 Proxies

When an user loads an entity in Hibernate and other ORM-softwares, the standard behaviour is to load a so called *proxy*, which is an object which is basically a copy of the loaded object, but with all fields set to null except for the ID. The reason why this is done is because we don't want to query the database for information that is not needed, but instead load specific fields for the entity when these are needed. This way, we can ensure that no unnecessary fields are loaded from the database, which could potentially harm the performance of the application. Proxies specifically plays an important role when we set the fetch type to *lazy*, which is something we will discuss more in detail later in the report.

1.1.1.3 Performance Tuning

ORM-tools have become widely popular due to the fact that they eliminate data access code in the application code. With high popularity, it is of great importance to find ways to overcome the issues and performance hogs that might exist in various ORM-software by default, usually called *Antipatterns*[7]. Performance optimizations with configurations in ORMs are usually carried out manually. The tweaking options in ORM:s are usually quite extensive but the amount of options differs depending on implementation. A few common techniques to tune performance are[20]:

- Customizing fetch type for associations
- Write native queries in complex situations
- Execute similar operations in batches instead of individual queries
- Use caching to minimize queries to the database

Whilst all of these are feasible options, the topic of investigation for this report will be focused around the first of the items in the list, *customizing fetch mode for associations*.

1.1.1.4 Fetch Modes

When an user loads an object, called a *root object* from here on, it will depend on the predefined fetch mode of that object and its associations what will initially load. In the case of Hibernate, the fetch mode for each association will have a default value depending on the cardinality of the relationship between associations, as of JPA standards. If the

user wants to override the default fetch mode for an association, this is possible through annotations in connection to the mapping of the association, as seen in Figure 1.5 where fetch mode lazy is being set for the association *customer*. It is also possible to prefetch on query level through the *Criteria-interface* in Hibernate.

```
1  @Entity
2  public class Employee {
3
4      @Id
5      @Column(name = "employee_id")
6      @GeneratedValue(strategy = GenerationType.AUTO)
7      private Long id;
8
9      @Column(name = "name")
10     private String name;
11
12     @JoinColumn(name = "supervisor_id")
13     @ManyToOne(cascade = { CascadeType.ALL })
14     private Employee supervisor;
15
16     @JoinColumn(name = "supervisor_id")
17     @OneToMany(cascade = { CascadeType.ALL }, fetch = FetchType.LAZY)
18     private Set<Customer> customers;
```

Figure 1.5: Mapping class in Hibernate with fetch mode set to lazy for an association

The two fetch modes for associations in ORM-softwares are:

- *Lazy Fetch mode*
- *Eager Fetch mode*

The lazy fetch strategy implies that you only load an object association, such as the address for an employee in the figure 1.3 example, when the user specifically uses that association in the code. Due to the fact that when using lazy loading we only load what is needed in the application, lazy fetch is usually the default option for association mappings in most ORM-softwares.

The eager fetch strategy on the other hand implies that all the associations will get loaded once the root object is loaded, meaning that the returned object will have all the fields instantiated immediately when loading. This is a more expensive fetch strategy, since more fields will be loaded. Hence, it is recommended to use this strategy only in certain situations where the user knows that all the fields will be used[21].

1.1.2 N + 1 Problem

In the earlier subsection we explained briefly how fetch strategies can be set up for individual associations, in order to load only the necessary information. The problem is to determine *what* and *when* something is necessary, since that might change throughout program execution. Fortunately, there is the possibility to modify prefetch strategies on individual queries with the criteria interface in Hibernate. However, the amount of modification can become cumbersome for the developer. Generally, it is advised that the default

fetch strategy should be lazy, and then modify to eager in places where the developer are sure that all the information from that specific association will be used[21]. In spite of that, developers have to be cautious with lazy loading, since it implies that the database will be queried for each of the lazily accessed fields in a program. Let's set up an example:

In Figure 1.6, we have a relationship the same as the one in Figure 1.5. The first part shows the iteration over the collection association *customers* in Java-code, and the second part shows the generated SQL-queries.

```
1    Set<Customer> customers = emp.getCustomers();
2    for (Customer customer : subordinates) {
3        String name = customer.getName();
4    }
```

```
1    SELECT * FROM Employee WHERE ...
2    SELECT * FROM Customer WHERE customer_id = 1
3    SELECT * FROM Customer WHERE customer_id = 2
4    ...
```

Figure 1.6: Iterating over an uninitialized collection, resulting in the $N + 1$ problem

Notice that the first query is fetching the collection of uninstantiated Customers from the Employee table, and then we generate extra queries fetching the name for every element in the collection, resulting in $N + 1$ executed queries in total. In a situation where we have network delay, each of the queries will spend *network delay* + *execution time*. This can heavily affect performance, especially in systems where the network delay is noticable[10]. Therefore, we want to monitor the occurrences of this problem in order to be able to resolve these issues. Fortunately, there are a few different strategies to handle this performance antipattern, and we will present some of the more common options here.

The first option is to set the fetch strategy of the association to *eager*, which implies that the whole collection of customers would get loaded together with the specific employee currently handled. In cases where this is not desired, another option is to *batch* queries, meaning that we would load multiple uninitialized proxies if one proxy is being accessed. This would reduce the number queries executed depending on how big the batch size would be set to. In figure 1.7 we set it to 25, and if the collection size is 50, Hibernate would generate two queries.

```
1    @JoinColumn(name = "supervisor_id")
2    @OneToMany(cascade = { CascadeType.ALL }, fetch = FetchType.LAZY)
3    @BatchSize(size=25)
4    private Set<Customer> customers;
```

Figure 1.7: Setting batch size for an association

It is also possible to use the Hibernate Query Language and the Hibernate Criteria-interface to define that specific associations should be loaded altogether. The Criteria-interface overrides the normal fetch strategy set for the association and sets it to the strategy provided in the criteria statement. An example of this can be seen in figure 1.8, where we

first use the *JOIN FETCH* directive to fetch and instantiate the collection association, and then in the second part we accomplish the same result by using the Criteria interface to override the default fetch type.

```
1  "from Employee employee join fetch employee.customers Employee"
```

```
1  Criteria criteria  = session.createCriteria(Employee.class);
2  criteria.setFetchMode("customers", FetchMode.EAGER);
```

Figure 1.8: Solving the $N + 1$ problem with *HQL* and *Criteria*-API in Hibernate

There are also other options in JPA 2.1, including the definition of *NamedEntityGraphs* and *DynamicEntityGraphs*. They let the user define graphs for entities that should be loaded from the database in certain scenarios. The user can define hints for each of the entity of this graph. An example of this technique can be seen in figure 1.9 where we add customers to the entity graph in line 2, and then use this graph when we load the employee in line 7 to also fetch the customers in the same query.

```
1  EntityGraph<Employee> graph =
   ↪  entityManager.createEntityGraph(Employee.class);
2  graph.addAttributeNodes("customers");
3
4  Map <String, Object> hints = new HashMap<String, Object>();
5  hints.put("javax.persistence.loadgraph", graph);
6
7  Employee employee = entityManager.find(Employee.class, 1L, hints);
```

Figure 1.9: Using a DynamicEntityGraph to fetch customers with the loaded Employee

1.1.3 ORM Optimization Difficulties

As shown in the earlier sections, there are numerous ways to improve the performance of systems using ORM-tools. However, these optimizations that are available all have to be carried out manually, which can be demanding from a technical aspect. The process of finding and maintaining the optimal configuration for software architectures using ORM-software is time-consuming and will require some in depth analysis of the system, in combination of wide knowledge of the different optimization techniques available in the ORM-tool. Not only that, but the fact that these settings only will affect performance make it more difficult from an implementation standpoint. When changes happen that might affect the viability of the current settings, the tuning has to be redone again. In conclusion, maintainability of the system will suffer.

Additionally, these optimizations tend to be error-prone, and when errors occur in these situations they can be the cause of critical performance losses[25]. Moreover, the loss of performance when making errors can be of a larger magnitude than if one were to leave the default configuration untouched, meaning that the optimization requires deep technical

understanding and needs to be tested thoroughly in order to grant better performance. To conclude, manual tuning of mappings in ORM-tools is complicated and simplification, or even elimination, of this process would lead to immense software engineering benefits, similar to the simplification of development that automatic memory management had when it was introduced.

In many companies that work with ORM-software, the problems mentioned above can compose time-consuming and error prone challenges. **itestra GmbH**, which is the host company for this research, is a software consulting company working with ORM-tools to a wide extent. Within their projects these issues have become apparent, and they now wish to come up with a solution.

1.1.4 Autofetch - the solution to manual prefetching

In the year of 2006, Ali Ibrahim and William Cook of the University of Texas developed a method to automate prefetching, based on earlier access statistics of associations categorized by program state[25]. According to their study, the technique performed identically to a hand tuned configuration and could eliminate up to 99.8% of the queries in the OO7-benchmark[22]. The technique has been implemented in two ORM-tools so far:

- Hibernate 3.1, stand-alone plug-in
- Ebean (from version 0.9.7 onwards), included in the software by default

However, this study was performed in 2006 and since then there has not been any significant research or follow up development on this topic. Neither has the Hibernate implementation been updated since the initial release. Thus, we have decided to migrate the old Autofetch-tool to the same version of Hibernate that a project of itestra GmbH uses. Moreover, we will also migrate the tool to the latest release of Hibernate[1] to have the possibility to test the tool on new projects in the future.

1.2 Motivation

With the absence of up to date versions and research of the Autofetch methodology, combined with itestra's experiences with prefetching in their enterprise-sized projects, the topic was chosen for this thesis with the goal to simplify the future endeavours to improve the performance of projects using ORMs. Not only will it save time for developers and potentially improve performance, but also make the system more maintainable since data mapping changes does not require reconfiguring of the prefetching modes with Autofetch enabled. According to itestra employees, the process of manual tuning within ORM:s are difficult to execute properly and the effects of each prefetching setting can be laborious to track. With potential performance improvements and these software engineering benefits in mind, the topic seemed promising.

Additionally, Autofetch was initially tested against benchmarks and other "dummy" projects. While this gives some indication of the performance of the methodology, it leaves some questions unanswered. Primarily, it does not answer how good of a fit Autofetch

[1] Hibernate 5.3.0.Final

is with Enterprise systems, such as systems based on *JavaEE*. In this study, we hope to answer some questions that Autofetch left unanswered in the original study, by evaluating the migrated tool against a web-based JavaEE-system.

In order to investigate the true characteristics of this methodology in a real world scenario, we have set up some questions that we want to answer with the research. The main questions to answer are presented in Table 1.1 and lay the foundation for the Contributions chapter.

Number	Question
1	Can Autofetch be migrated to the latest release of Hibernate?
2	How does Autofetch affect performance in a JavaEE application?
3	Does Autofetch simplify the process of optimizing ORM-configurations?

Table 1.1: The research questions that we strive to answer in this report

When answering these questions, we hope to give an approximation of what to expect from the technology itself, but also the capabilities and faults of the developed implementation.

1.3 Related Work

Autofetch and Hibernate are based on the concept of *persistence*. In the topic of persistence in Java, there has been numerous studies. Moss, J. Eliot B., and Antony L. Hosking[34] propose ideas to integrate persistence mechanisms in Java, and discuss the problems relating persisting more than just code, but also the virtual machine specification and libraries, creating a situation far more complex than earlier attempts with other programming languages. Jordan, Mick J., and Malcolm P. Atkinson[2] establish that in order for *Orthogonal Persistence* to be possible, it needs to be applied to the entire Java-platform. They also discuss the problems with the Orthogonal Persistence approach and what needs to be done in order for it to be viable on a larger scale, including support for threads, and make it scale the same way as standard Java environments.

Persistence object oriented programming languages lead us up to the topic of the *impedance mismatch problem* which Ambler, Scott W. explains from a more practical standpoint[4]. This study focuses on providing solutions for implementation of object-relational mapping, laying a foundation for many implementations of the *Object-Relational Mapping* concept, including Hibernate. Ireland, Christopher, and David Bowers on the other hand, focuses solely on the theoretical aspects of the impedence mismatch problem, providing arguments for why this problem is a wicked one, meaning that the problem is not solvable due to its many-sided nature[5]. Each attempt of solving the problem will cause a different problem to arise.

The concept of ORM has been a topic of massive debate and has led to numerous effort trying to evaluate the method's viability. The opinions on this matter change from time to time depending on the current trends and the wide range of opinions on the matter tend to influence the research. Joshi, Aditya, and Sanjeev Kukreti[24] for example compare ORM-tools to traditional database access techniques in terms of performance, code

readability and maintenance. They reach the conclusion that in most cases, ORM-tools are worth considering despite the performance overhead, due to the numerous benefits in terms of readability and maintainability that it offers. Lascano, Jorge Edison[30] reach similar conclusions in a paper where JDBC is compared with JPA-implementations, referring to the scalability that ORM-tools offer. Ghandeharizadeh, Shahram, and Ankit Mutha[11] conduct similar research but in a social network setting, and reaches the conclusion that the JDBC-implementation offers better performance, however only because the instance of *N + 1* problem that appears in the Hibernate equivalent. Eliminating this instance results in identical performance between the two platforms. Alvarez-Eraso, Danny Alejandro, and Fernando Arango-Isaza perform a performance study in a web-application environment where Hibernate is used together with the popular framework *Spring*. The results show that the performance of ORMs is similar to the performance of non-optimized handwritten queries, showing a weakness in performance with ORMs in complex situations.

Others are more critical of the concept of ORM-tools, for example Neward, Tom[29] who dedicates an entire article to criticize the development of ORM-tools, with the argument that meanwhile ORM-tools have good intentions, it only creates a more complex problem as time passes. He proposes solutions, for example the abandonment of objects, change from relational storage, integration of relational concepts in programming languages etc.

The wide usage of ORM-tools has led up to studies on the consequences of using ORM-tools and the correct usage of these to reach optimal performance and maintainability. Chen, Tse-Hsun, et al. investigate how ORM-tools and its code can be maintained within an application[8]. Surprisingly, the results show that ORM-code tend to change more frequently compared to other code and the changes are usually more scattered and complex. This indicates that the usage of ORM-tools can make maintainability suffer in systems. The same authors, Chen, Tse-Hsun, et al[7] also create a method for finding so called *Antipatterns* in ORM-code. This includes the detection of instances of the *N + 1*-problem, unnecessary Eager loading to mention a few examples. This is a slightly different approach from that of Autofetch, due to both detection and resolving is carried out manually, compared to Autofetch where all these antipatterns are resolved automatically. Aoki, Yasuhiro, and Sigeru Chiba[37] implemented an aspect oriented programming-based approach which lets developers add dynamic contexts for prefetching to improve the performance of a system.

In the topic of prefetching in persistent architectures, there has been some earlier research. Ramachandra, Karthik, and S. Sudarshan[36] come up with an approach to holistically optimize prefetching. However, the approach is different from the method found in this report in the way that it automatically rewrites application code in order to accomplish efficient prefetching, compared to Autofetch that uses the earlier execution statistics on associations to concatenate prefetch directives to queries. The performance gain using this method is according to the authors approximately 50% compared to a non optimized configuration. There are also studies about a more specific problem when deciding what to prefetch, for example answering the question *how to determine Access patterns in persistent systems?* Touma, Rizkallah, et al[40] come up with a technique to predict access patterns before program execution using static code analysis. This means that the technique does not add any overhead to the execution of the program, which is an advantage compared to methods such as Autofetch. Garbatov, Stoyan[39] present methods for data

access patterns through the use of three powerful tools *Bayesian Inference*, *Importance Analysis* and *Markov Chains*. All three models result in high precision, however with noticeable overhead (5-9%).

Ibrahim, Ali, and William R. Cook[25] laid the foundation for this research by presenting the theory and implementing a tool. The original article covers more of the theoretical aspects of Autofetch, whereas this study more focus on the practical side of Autofetch in a real world scenario, together with the details of migration of the old tool. Most of the theoretical models and implementation details remain the same as in the original article.

1.4 Contributions

Ibrahim & Cook developed Autofetch for Hibernate in 2008 and they claim to have reduced the number of queries in their test applications by up to 99.8%. They tested the tool on benchmarks such as *TORPEDO*[45] and *OO7*[22]. Since their article, there has not been any follow up research on this specific topic.

In this work we will migrate the developed Hibernate tool and investigate how the tool works in practice and how it affects the performance of a JavaEE application. With the presence of the N + 1 problems in ORM-software and its effect on performance in systems, we are investigating whether Autofetch can be applied to a JavaEE project to eliminate queries to the same extent as a manually optimized configuration of the same project, but we are also investigating if Autofetch improves performance compared to a non optimized version of the project. The contributions of this work are the following:

- Migrate Autofetch from Hibernate 3.1 to Hibernate 4.3
- Evaluate Autofetch in terms of practicality and gained performance on a JavaEE application

1.5 Disposition

In this section we will present a brief overview of the different sections of the report.

- *Chapter 1 Introduction:* Introduction to the specific field of research, together with the problems that we want to answer in our research. It presents a thorough background to give the reader insight why Automatic prefetching is a concept that can relieve the developers of the mundane task of adding manual prefetch hints.
- *Chapter 2 Approach:* Explains the general theory and implementation of Autofetch, based on the original article by Ibrahim & Cook, together with the implementational changes with the migrated versions. Finally, the method of evaluation is presented.
- *Chapter 3 Evaluation:* Presents the findings of the tests carried out in the project. Covers the performance of the tool in a JavaEE-application and a discussion about the findings.

- *Chapter 4 Conclusion:* Answers the research questions and proposes future work together with faults of this research. Finally, the questions in table 1.1 will be answered.

Chapter 2
Approach

In this section, we will present the theoretical foundation that the implementation is based on together with the implementation details. Furthermore, the method of evaluation will be presented. The first section 2.1 is solely constitutes a summary of the original tool of Ibrahim & Cook and does not contain any contributions from this author. Instead, the contributions of this work start at section 2.2 and continues until the end of the report.

2.1 Original Tool

The initial problem of the study, in which Autofetch was developed, was addressed by the original authors of the tool, Ali Ibrahim and William R. Cook. The models of profiling traversals, query classifications and predicting traversals presented in this report are identical to the models in the original report by Ibrahim and Cook[25]. Additionally, the internal implementation structure remain mostly the same, with a few exceptions. Therefore, it is important to note that this work mostly consist of the migration to the new versions of the platform of Hibernate together with a evaluation of the tool and the methodology itself. Additionally, we will evaluate whether Autofetch is applicable to a JavaEE system in terms of ease of usage and gained performance. We will now present the main structure of the old tool, the migrated tool and the design choices made in the new implementation.

2.1.1 Overview

In this subsection we will summarize the most important aspects of Autofetch. The summary is based on the third section in the original Autofetch report[25]. Before going into detail about this topic, it is important to introduce the main models of Autofetch. There are three aspects of the Autofetch that are of particular importance, since they propose solutions to three thoroughly researched topics. The three main models answer the questions found in Table 2.1.

Number	Question
1	How do we do profiling to get user statistics?
2	How do we classify queries?
3	How do we traverse associations?

Table 2.1: The questions leading up to the models

As mentioned earlier, Autofetch strives to eliminate the cumbersome process of manually adding prefetch directives to the mapping file of an ORM-software. Instead, the usage of an association of objects in a certain program state should be tracked, so that in future iterations of that program state the statistics of earlier iterations indicate whether the association is necessary to prefetch in that specific program state or not.

The authors of the original Autofetch article defined two concepts; The type graph and the object graph. The type graph defines the relations between the different object types, similar to a *Entity Relationship Diagram*. The object graph on the other hand represents the objects of a type defined in the type graph. The two graphs are shown in Figure 2.1 and Figure 2.2 respectively. In Figure 2.2 we see the root object *Company A* and its associations *Employee A* and *Employee B*. The fraction seen above the associations is the probability $\frac{used}{potential}$, meaning the probability that we load this association. With these two concepts defined, we can describe the method with further detail.

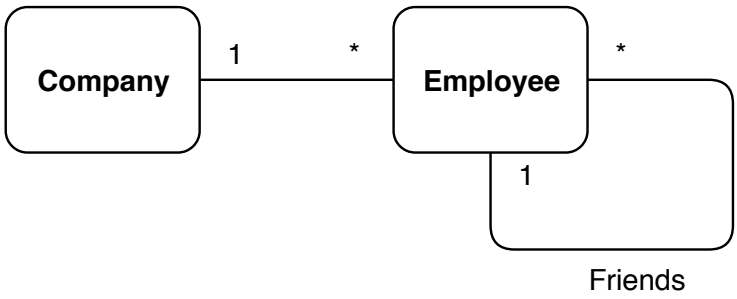

Figure 2.1: Simple type graph describing the relationship between entity types

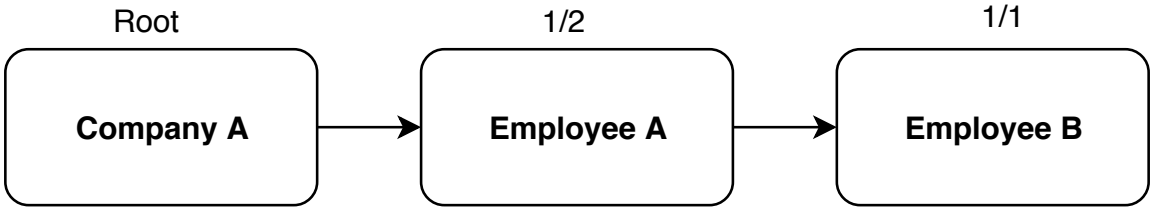

Figure 2.2: Object graph with root object and fetch probability marked

Traversals are the accessed object and associations used from the result of a query. These traversals are then merged to *Traversal Profiles*, which include the statistics about the associations rate of access. Furthermore, the queries are also classified into *Query Classes*, which is a classification of queries that are likely to have similar traversal profiles.

For each of these query classifications, query prefetch directive will be added to the queries of these classes based on the traversal profile statistics. This results in automated prefetching. The chain of events constituting the Autofetch methodology can be seen in Figure 2.3, where we start from the point where the user loads an entity from the database in Hibernate. Before the entity is being loaded, we check if the entity has any existing prefetch paths associated with it. If we have that, we prefetch each of the prefetch paths that has a probability of being loaded above a certain threshold, in this case set to 0.50. When we load the entity, we add trackers to the entity that will increment statistics accordingly, so that we can check the fetch probability for this entity in future executions of the program.

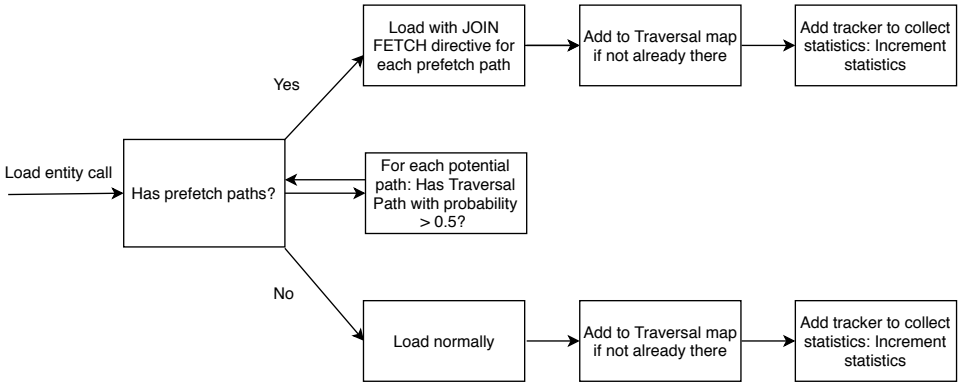

Figure 2.3: Program flow graph when a user loads an entity using Autofetch

2.1.2 Models

In the earlier section, the general methodology was explained. However, there are models that constitute the foundation for Autofetch's mechanisms that will need to be defined in order resolve the questions in table 2.1. Each of the questions in 2.1 refers to each subsection in this section, for example question one refers to subsection 2.1.2.1. This subsection will act as a summary for the models presented by Ibrahim and Cook in the original Autofetch article[25].

2.1.2.1 Profiling Traversals

Profiling is used in Autofetch to determine what parts of the database that are used in certain operations. In Autofetch, five sub models were defined; *Type*, *objects*, *queries*, *traversals* and *traversal profiles*.

Type Graph: Let T be the finite set of type names and F be the finite set of field names. A type graph is a directed graph $G_T = (T, A)$

- T is a set of types.
- A is a partial function $T \times F \longrightarrow T \times \{single, collection\}$ representing a set of associations between types. Given types t and t' and field f if $A(t, f) = (t', m)$ then there is an association from t to t' with name f and cardinality m where m indicates whether the association is a single- or multivalued association.

Object Graph: Let O be the finite set of object names. An object graph is a directed graph G_O = $(O, E, G_T = (T, A), Type)$. G_T is a type graph and $Type$ is a unary function that maps objects to types. The following constraints must be satisfied in the object graph G_O:

- O represents a set of objects.
- $Type : O \longrightarrow T$. The type of each object in the object graph must exist in the type graph.
- $E : O \times F \longrightarrow powerset(O)$, the edges in the graph are a partial function from an object and field to a set of target objects.
- $\forall o, f : E(o, f) = S$

  $A(Type(o), f)$ = (T', m)

  $\forall o' \in S, Type(o') = T'$.

  if m = $single, then\ |S| = 1$.

The edges in the object graph corresponds to the edges in the type graph. Single associations will have one target object, and multivalued associations will have an empty or non-empty collection as target object. The reason that empty collections are represented in the object graph is that the ORM have to query the database in order to know whether a collection is empty or not i.e. it does not know anything about the cardinality of the collection. This is not the case with single value associations, due to a special representation of null value targets.

Queries: A query is defined as a function that returns a subgraph of the whole database graph. There are two properties that Ibrahim and Cook call *extent type* and *criteria*. Extent type is defined as the type of root object that is returned by the query, and criteria is conditions that need to be fulfilled in order to return that specific extent type. The only preconditions regarding the query language is that it have to have a object-oriented view of persistent data. Additionally, it must be possible to add prefetch directives based on the extent type. Noteworthy about queries is its first-class value nature, similarly to how functions work in functional programming languages[25].

Traversals: The query returns an object graph that is a subgraph of the database, and the question is how this subgraph should be traversed. Ibrahim and Cook define traversals with the following model:

A traversal is represented as a forest where each tree's root is a root object in the result of a query and each tree is a sub graph of the entire object graph. Let R denote a single tree from the traversal on the object graph G_O = (O, E).

$R = O \times (F \longrightarrow \{R\})$ where $(o, (f, r)) \in R$ implies $|E(o, f)|$ = $|r|$ The only associations that are being put into a traversal are the ones that will lead to a database load. There are three cases in which a navigation of an association does not lead to a select query:

- Null value single association
- Association was already cached earlier in the program
- There already exist a path to that association with smaller distance

If an empty collection association is being traversed, it will be queried and be put in the traversal.

Traversal Profiles: A traversal profile is an aggregation of the traversals a group of queries. Combining these in a specific manner creates a pattern of traversal for a category of queries.

$P = T \times N \times N \times (F \longrightarrow P)$ such that for all $(t, used, potential, (f, p)) \in P$:

1. $A(t, f)$ is defined
2. $used \leq potential$.

The nodes contain information on how many times the node has been loaded from the database, which is refered to as *used* from here on, and how many times the node has had the chance to be loaded, referred to as *potential* from here on. To clarify, *potential* can be defined as how many times the program has had a direct reference to this node without loading it.

2.1.2.2 Query Classification

In the earlier chapter we introduced that Autofetch somehow groups certain queries together in the traversal profile stage of the method. The hypothesis the authors of the original Autofetch article had was that using a combination of the line number and stack trace to classify queries would lead to an accurate mapping. The reasoning behind this was that the stacktrace shows the flow of the application. Other options such as using the query string to classify queries was discarded due to the imprecision that would lead to. This is due to the fact that the same query can be used in completely different scenarios, and in these different scenarios different associations might be relevant[25].

2.1.2.3 Predicting Traversals

The last of the models presented by the authors of the original Autofetch article is regarding how to properly predict how the program will traverse in the future. Having built the earlier models with profiling makes this model straight forward. The goal is to have the traversal only traverse the resulting query object graph, so that no additional select query has to be executed. In this way, we can minimize the amount of round trips to the database. Autofetch uses a user defined threshold which is used the comparison with the $\frac{used}{potential}$ fraction to determine whether to prefetch the node. However, the node should only be prefetched if the parent node is prefetched. Hence the formula will be the following:

- $f(n) = (used(n)/potential(n)) * f(p(n))$
  - where $f(root) = 1$
  - $f(n) \leq f(p(n))$

The calculation of probabilities is done through a depth first traversal. This ensures that the calculation is carried out once per node.

2.1.3 Implementation

In this section, we will summarize the implementation by the authors of the original Autofetch article. Autofetch was originally developed for Hibernate 3.0 by Ibrahim and Cook and then migrated to 3.1. In the first version, the functionality of multiple collection prefetching was not implemented which reduced the potential performance improvements. It was when the 3.1 version was released the potential of the tool increased due to the inclusion of multiple collection prefetching[25]. This version had a codebase size of 7288 lines of code.

Autofetch's implementation partly replaced existing classes in the Hibernate library and wrapped it or replaced it with classes with modified behaviour. Consequently, there is a lot of duplicated code in the old version of Autofetch simply due to the reason that they could not build the design however they wanted, but instead had to adapt to the static design of the Hibernate version of the time. In general, we think the implementation suffered from readability and maintainability issues due to the workarounds needed to implement it initially.

2.1.3.1 Configuration and Integration

Integrating the first version of Autofetch was done through manual download of a Autofetch-zipfile, containing the code and an Ant-script to create the two necessary JAR-files. These two JAR-files were then added to the class-path. With these two JAR-files added to the project, the specific instance of *AutofetchConfiguration* could be instantiated, allowing the user to enable automatic tuning. In normal Hibernate, the user would instantiate *Configuration* instead.

Hibernate uses event listeners in order to track and intercept events. For example, there are certain types of listeners for when the user loads entities using the *load*-method in Hibernate. These had to be replaced with custom versions in order to inject the modified load mechanism used in Autofetch, which was shown in Figure 2.3.

2.1.3.2 Added/Modified classes

In order to add the Autofetch-functionality, the authors of the first Autofetch implementation had to add new and extended classes of the Hibernate framework in order to get Autofetch to function as intended. In Table 2.2 we display all the wrapper classes that was needed in order to make Autofetch run on Hibernate. Moreover, in Table 2.3 we display the entirely new classes added by the authors of the old tool.

Class	Class name
1	AutofetchConfiguration
2	AutofetchProxyFactory
3	AutofetchInterceptor
4	AutofetchInstantiator
5	AutofetchHbmBinder
6	Collection Wrapper classes
7	AutofetchCriteria

Table 2.2: Custom Autofetch wrappers of default Hibernate classes in Hibernate 3.1

Class	Class name
1	Extentmanager
2	Path
3	ProgramStack
4	Property
5	TrackableEntity
6	Trackable
7	TraversalProfile
8	Statistics

Table 2.3: Custom classes of Autofetch in Hibernate 3.1

2.1.3.3 Bootstrapping

In order to add Autofetch functionality to Hibernate, we need to be able to track the various entities of the database. Therefore, we need to somehow mark the fetched objects with some type of marker that keeps track of when the entity is being accessible and when the entity is actually being accessed by the root object of the query so that we can collect the necessary statistics. This can be done in Hibernate by altering the bootstrapping process, which is what the authors of the first tool did.

They modified the bootstrapping process of Autofetch, so that we instantiate entities using proxies so that all the accesses to these objects can be tracked. Also, the internal proxy factory is set to be the AutofetchProxyFactory, which adds the *TrackableEntity* interface to each created proxy, which can be seen in line 6 of Figure 2.4 This allows the proxy to be tracked.

```
1  public static Class getProxyFactory(Class persistentClass,
2  String idMethodName) {
3      if (!entityFactoryMap.containsKey(persistentClass)) {
4          Enhancer e = new Enhancer();
5          e.setSuperclass(persistentClass);
6          e.setInterfaces(new Class[] { TrackableEntity.class });
7          e.setCallbackTypes(new Class[] { MethodInterceptor.class,S
8              NoOp.class, });
9          e.setCallbackFilter(new EntityCallbackFilter(idMethodName));
10         e.setUseFactory(false);
11         e.setInterceptDuringConstruction(false);
12         entityFactoryMap.put(persistentClass, e.createClass());
13     }
14
15     return entityFactoryMap.get(persistentClass);
16 }
```

Figure 2.4: Adding the interface to the entities in EntityProxyFactory during bootstrap

2.1.3.4 Entity Mapping

In Hibernate 3, the Configuration-class only handled entity mappings in Hbm-format, which is a specific xml-file used for entity mapping. At this time, annotation based mapping, which means that entities are mapped in the code, was not supported. In order to use annotation based mapping, the *AnnotationConfiguration* had to be used. The problem was that this type of configuration was not supported by Autofetch, leaving the user without options when it comes to entity mapping.

The central class in the mapping handling was AutofetchHbmBinder, which parsed the mapping-files to create the configuration time metamodel. This class also handled the wrapping of the native collection types in Hibernate to enable tracking of operations made on the collections. An example of the custom collection wrappers that the HbmBinder wrapped collections with can be seen in Figure 2.5, where we in line 6 increment the used statistic when calling the *size*-method.

```
1  public class AutofetchSet extends PersistentSet implements Trackable {
2
3      @Override
4      public int size() {
5          int ret = super.size();
6          this.accessed();
7          return ret;
8      }
```

Figure 2.5: An example collection wrapper used to be able to track accesses to the persistent collections

2.1.3.5 Proxy Creation

Since Autofetch is dependent on using custom proxies in order to inject its statistics mechanism when loading objects, this had to be altered by the authors of the old tool.

The general structure of the creation of proxies is according to the *Proxy pattern*, similar to the one used natively by Hibernate. In Autofetch, the pattern is extended by a class *EntityProxyFactory* which uses a custom method interceptor called *EntityProxyCallBack*. *EntityProxyFactory* functions as a proxy provider for each persistent entity, holding a map with each entity and its respective factory. The *EntityProxyCallback* intercepts the calls to the proxy to enable/disable tracking and various methods for handling the entity statistics.

2.1.3.6 Listeners

During runtime of an application using Autofetch, the *AutofetchLoadListener* or *InitializeCollectionListeners* will be called, intercepting the normal method calls with the custom operations of Autofetch when the user does certain operations such as loading or initializing a collection. It does this in order to check for prefetch directives for this entity generated by earlier iteration of Autofetch and to update the tracking. We can see this in Figure 2.6, where we in the loadFromDatasource-method check for prefetch paths for this entity, and then modify the result depending on the statistics of the prefetch paths seen between line 11 and 18 in Figure 2.7.

```
1  @Override
2  protected Object loadFromDatasource(LoadEvent event,
3  EntityPersister entityPersister, EntityKey entityKey,
4  LoadType loadType) throws HibernateException {
5
6      String classname = entityPersister.getEntityName();
7      if (log.isDebugEnabled()) {
8          log.debug("Entity id: " + event.getEntityId());
9      }
10     List<Path> prefetchPaths = extentManager.getPrefetchPaths(classname);
11     Object result = null;
12     if (!prefetchPaths.isEmpty()) {
13         result = getResult(prefetchPaths, classname, event.getEntityId(),
14         event.getLockMode(), event.getSession());
15         if (result instanceof HibernateProxy) {
16             HibernateProxy proxy = (HibernateProxy) result;
17             if (proxy.getHibernateLazyInitializer().isUninitialized()) {
18                 throw new IllegalStateException("proxy uninitialized");
19             }
20             result = proxy.getHibernateLazyInitializer()
21             .getImplementation();
22         }
23     } else {
24         result = super.loadFromDatasource(event, entityPersister,
25         entityKey, loadType);
26     }
27     extentManager.markAsRoot(result, classname);
28     return result;
29 }
```

Figure 2.6: The overidden loadFromDatasource-method used in AutofetchLoadListener

2.1.3.7 Modification of Select Queries

One of the advantages of using a persistence layer like Hibernate is that the user can use normal Java-syntax and let the used framework handle the database calls. With Autofetch,

the loaded root-object and its associations will be loaded using the overriden loadFromDatasouce-method in Figure 2.6. Depending on if this entity has prefetch paths associated with it, a method called getResult will be called instead of the normal loadFromDatasource-method. In this method, seen in Figure 2.7, we will append the specific prefetch path for this entity if the probability that that entity will be traversed is higher than the set threshold.

Here the JPQL specific JOIN FETCH directive is used to tell Hibernate to load and initialize the appended element. In this way, we do not have to load the association separately with an independent query, and therefore we can decrease the amount of executed queries. In many cases, lower amount of executed queries means better performance of the application, especially when network delay is a factor. This is the last mechanism in the chain of events of the Autofetch tool and is ultimately the mechanism that makes Autofetch tune the queries generated by the underlying application.

```
1  public static Object getResult(List<Path> prefetchPaths, String classname,
2  Serializable id, LockMode lm, Session sess) {
3      StringBuilder queryStr = new StringBuilder();
4      queryStr.append("from ").append(classname).append(" entity");
5      Map<Path, String> pathAliases = new HashMap<>();
6      int aliasCnt = 0;
7      pathAliases.put(new Path(), "entity");
8
9      // Assumes prefetchPaths is ordered such larger paths appear after smaller
       ↪  ones.
10     // Also assumes all prefixes of a path are present except the empty prefix.
11     for (Path p : prefetchPaths) {
12         String oldAlias = pathAliases.get(p.removeLastTraversal());
13         String newAlias = "af" + (aliasCnt++);
14         String lastField = p.traversals().get(p.size() - 1);
15         pathAliases.put(p, newAlias);
16         queryStr.append(" left outer join fetch ");
17         queryStr.append(oldAlias).append(".").append(lastField).append("
           ↪  ").append(newAlias);
18     }
19     queryStr.append(" where entity.id = :id");
20
21     if (log.isDebugEnabled()) {
22         log.debug("Autofetched Query: " + queryStr);
23     }
24
25     Query q = sess.createQuery(queryStr.toString());
26     q.setLockMode("entity", lm);
27     q.setFlushMode(FlushMode.MANUAL);
28     q.setParameter("id", id);
29
30     long startTimeMillis = System.currentTimeMillis();
31     Object o = q.uniqueResult();
32     if (log.isDebugEnabled()) {
33         log.debug("Query execution time: " +
34         (System.currentTimeMillis() - startTimeMillis));
35     }
36     return o;
37 }
38 }
```

Figure 2.7: The getResult-method in AutofetchLoadListener which appends the JOIN FETCH directive to the select query

2.2 Migration of Autofetch

Autofetch was released in 2008 for Hibernate 3.1, and since then the methodology has not been updated. By the time of this report, the latest release of Hibernate is 5.3.0.Final. Between these releases, there has been a lot of changes to the framework, such as the change from JPA 2.0 to JPA 2.1, introducing many new features. The Hibernate core has been thoroughly refactored and concepts such as *Services* has been introduced. Therefore, the tool needed to be adjusted to these changes and new features in order to function on newer version of the framework. Additionally, since the target project of this evaluation uses Hibernate 4.3, we also needed to migrate a version to this release.

In order to achieve faster development pace, the migrated versions was decided to be open source based, meaning that anyone could work on the migration. This was necessary due to the big amount of changes needed to be done in order to get the tool to a working state. The project was hosted on Github[1] in order to increase visibility to the project. This allowed other developers to contribute to the effort. Furthermore, since Hibernate itself and the original tool was under LGPL 2.1, it was decided that the continuation of the tool's development should be under the same license. With the Hibernate documentation being incomplete and the framework itself being under constant development, receiving tips and implementation help was needed.

We initially planned to perform the migration through incremental upgrades of the Hibernate core to find the incompatibilities, in an effort to gradually perform changes to tool. However, this proved to not be an efficient method of migration, since an architectural change in one version not necessarily remained in the next version which lead to that many time-consuming changes ended up not being used. Luckily, we realized this mistake quickly, and therefore not a lot of time ended up being wasted. Instead, the target version was used immediately and the changes could be applied to the original code until it reached a state where it was executable again, but for the new version. During the migration process, the architecture of the tool had to undergo some changes in order for the tool to work. In the next section, the major changes will be presented. Smaller miscellaneous fixes have been omitted from the report for relevance reasons.

2.2.1 Implementational Differences

Hibernate as a framework has been changed immensely from version 3 to version 5. Some classes and patterns from Hibernate 3 was changed or deleted completely in the newer versions. This forced some changes to be implemented in order for the tool to function. Here we will present some architectural changes made. In general, the changes mostly affect the classes in table 2.3, which goes in line with the general scope of migrating and evaluating the existing tool, rather than creating a completely new implementation. We did most of the work regarding the migration with the help of the migration guides on the Hibernate website[31]. The total size of the new project ended up being 5189 lines of code, compared to 7288 lines of code in the original tool. This effort was evenly distributed between the author and one other contributor on Github. The whole migration effort lasted around four months of time.

[1] https://github.com/ErikJonasson/Autofetch

2.2.1.1 Configuration and Integration

The original tool was packaged as an Ant-script that created two JARs which the user added to the classpath of the desired project. This comes with one disadvantage, namely that the user has to manage the dependencies. In the migration we wanted to streamline this process, so we decided to utilize *Maven* for handling the dependency management. It also adds the simplicity for users to integrate the tool by simply adding the Autofetch dependency to the *pom.xml*-file. Lastly, it adds the option to easily add new plugins to the configuration, such as *Bytecode Enhancement*, which we will explain more in detail in section 2.2.1.6.

With the evolution of Hibernate, the ways of integration of user modifications have changed throughout the releases. Hibernate uses so called *tuplizers* to manage the representation of data depending on what type of entity type that entity has. Hibernate also lets users create their own custom tuplizers in order to modify the various behaviours of Hibernate. With Hibernate 3, in order to add a custom tuplizer, you had to either add it manually in the mapping Hbm.xml-file or as in the case of the original Autofetch version, go into the core classes of Hibernate, make a custom copy of this class, add the tuplizer programmatically, and then find a way to inject this modified class. The custom tuplizer is needed for Autofetch due to the fact that we need to toggle the tracking on and off in different scenarios.

The problem with the first approach regarding adding custom tuplizers is that the user of the tool has to go through a longer configuration process in order for the tool to function, which we want to keep to a minimum in order to increase usability for the user. The latter approach creates a duplicated code problem with maintainability issues. In the next version the functions may change, and then the current version of the tool will not work. These two approaches can be observed in Figure 2.8 and 2.9 respectively.

```
1  <hibernate-mapping>
2  <class entity-name="Employee">
3
4  <tuplizer entity-mode="POJO"
5  class="AutofetchTuplizer"/>
6
7  <id name="id" type="long" column="ID">
8  <generator class="sequence"/>
9  </id>
10
11 <!-- other properties -->
12 ...
13 </class>
14 </hibernate-mapping>
```

Figure 2.8: Adding a custom tuplizer manually for an entity in Hibernate 3

```
1  private static void bindPojoRepresentation(Element node,
2  PersistentClass entity, Mappings mappings, java.util.Map metaTags) {
3
4      String className = getClassName(node.attribute("name"), mappings);
5      String proxyName = getClassName(node.attribute("proxy"), mappings);
6
7      entity.setClassName(className);
8
9      if (proxyName != null) {
10         entity.setProxyInterfaceName(proxyName);
11         entity.setLazy(true);
12     } else if (entity.isLazy()) {
13         entity.setProxyInterfaceName(className);
14     }
         ⋮
15     // Element tuplizer = locateTuplizerDefinition( node, EntityMode.POJO );
16     // AHI: inject our tuplizer class here.
17     entity.addTuplizer(EntityMode.POJO,
18     "org.autofetch.hibernate.AutofetchTuplizer");
19 }
20
```

Figure 2.9: Adding the AutofetchTuplizer programmatically in the modified HbmBinder-class

In Hibernate 4, the concept of *Services* was introduced. This allowed for user manipulation in an easier way, since users can extend existing services, or create their own services which can then be injected[33]. Services made it possible to instead of ”hacking” the hibernate core, a service could be created independently of the core, leading to a less rigid and fragile design. This also changed the way custom listeners are added by the user, from adding them in the configuration-class to having to inject them using a special service during bootstrap. The change on the integration of listeners forced us to create a new integration method for the custom listeners that are needed for Autofetch to work. This will be described in detail in section 2.2.1.3.

In the 4.3-version of the tool, a so called *Integrator* is used to inject a custom service called *AutofetchService*. The integrator gets injected by defining a source-file in the META-INF\services-folder, which is then parsed during the bootstrapping process of Hibernate. During the bootstrapping, all the various integrators will be integrated. The Integrator of Autofetch adds an *Initiator*, whose implementation can be observed in Figure 2.10. It instantiates the AutofetchServiceImpl, which acts as a singleton for the Extentmanager, managing all the main functionality of Autofetch. The AutofetchServiceImpl-class can be observed in Figure 2.11, with which the Extentmanager can be accessed.

Furthermore, the Integrator is responsible to add the Collection wrappers for the persistent collection types of Hibernate, such as the PersistentSet and PersistentBag. Lastly, it adds the custom listeners of Autofetch to allow tracking of entities.

We implemented the integration of Autofetch in the 5.3-version similarly to the implementation in the 4.3-version, but instead of adding the service through the Integrator, it is injected through the new *ServiceContributor*-interface.

```
final class AutofetchServiceInitiator implements
↪  StandardServiceInitiator<AutofetchService> {

    static final AutofetchServiceInitiator INSTANCE = new
    ↪  AutofetchServiceInitiator();

    @Override
    public AutofetchService initiateService(Map configurationValues,
    ↪  ServiceRegistryImplementor registry) {
        return new AutofetchServiceImpl();
    }

    @Override
    public Class<AutofetchService> getServiceInitiated() {
        return AutofetchService.class;
    }
}
```

Figure 2.10: The AutofetchServiceInitiator added by the Integrator

```
final class AutofetchServiceImpl implements AutofetchService {

    private final ExtentManager extentManager;

    AutofetchServiceImpl() {
        this.extentManager = new ExtentManager();
    }

    @Override
    public ExtentManager getExtentManager() {
        return extentManager;
    }
}
```

Figure 2.11: The custom service managing the ExtentManager

2.2.1.2 Collection Wrapping

In order for Autofetch to be able to do its tracking, the custom collection classes need to be used. The original tool integrated the wrappers of Autofetch in the custom class called AutofetchHbmBinder, which is a class that has been completely removed from Autofetch in the 4.3 version due to Hbm-mapping being deprecated in newer Hibernate releases. Instead the wrappers are integrated into Hibernate with the use of the Integrator, which during the bootstrap process iterates over the available properties and sets the property to the equivalent AutofetchCollection-type. The method handling the replacement of the collections can be observed in Figure 2.12. We implemented this to ensure automatic wrapping of collections without any user specification while keeping a maintainable design and eliminating the duplicated code that was used in the earlier AutofetchHbmBinder-class. The integration of the AutofetchTuplizer for each entity is done analogously. Furthermore, the AutofetchCollectionTypes now implement the UserCollectionType instead of extending CollectionType due to incompatibility between the CollectionType and the new implementation.

```
1  private static void replaceCollection(org.hibernate.mapping.Property
   ↪  collectionProperty, PersistentClass owner) {
2      if (!(collectionProperty.getValue() instanceof
       ↪  org.hibernate.mapping.Collection)) {
3          return;
4      }
5      org.hibernate.mapping.Collection value = (org.hibernate.mapping.Collection)
       ↪  collectionProperty.getValue();
6
7      if (value instanceof org.hibernate.mapping.Bag) {
8          value.setTypeName(AutofetchBagType.class.getName());
9      } else if (value instanceof org.hibernate.mapping.IdentifierBag) {
10         value.setTypeName(AutofetchIdBagType.class.getName());
11     } else if (value instanceof org.hibernate.mapping.List) {
12         value.setTypeName(AutofetchListType.class.getName());
13     } else if (value instanceof org.hibernate.mapping.Set) {
14         value.setTypeName(AutofetchSetType.class.getName());
15     } else {
16         throw new UnsupportedOperationException("Collection type not supported:
           ↪  " + value.getClass());
17     }
18 }
```

Figure 2.12: Wrapping the default CollectionTypes with the Autofetch equivalents

2.2.1.3 Listeners

The tracking of entities in Autofetch is based on the ability to listen to certain operations done by the user, such as loading an object. As mentioned in section 2.1.3.6, this is done through *Listeners*. Originally, custom listeners were added through the Configuration-class. However, in the *4.0*-version of Hibernate, the Hibernate team decided to create a specific ServiceRegistry for modifying and adding listeners called the EventListenerRegistry. This can be accessed during the bootstrap-process to add the desired listeners.

For the migrated versions of Autofetch, we used the EventListenerRegistry to implement a method to automate the integration of the custom listeners in the AutofetchIntegrator-class. The implementation is shown in Figure 2.13, where the custom load listener is being added in line 5 and the initialize collection listener is added in lines 7-8.

```
1  private void doIntegrate(ServiceRegistry serviceRegistry) {
2      final ExtentManager extentManager =
       ↪  serviceRegistry.getService(AutofetchService.class).getExtentManager();
3
4      EventListenerRegistry eventListenerRegistry =
       ↪  serviceRegistry.getService(EventListenerRegistry.class);
5      eventListenerRegistry.setListeners(EventType.LOAD, new
       ↪  AutofetchLoadListener(extentManager));
6
7      eventListenerRegistry.setListeners(EventType.INIT_COLLECTION,
8      new AutofetchInitializeCollectionListener(extentManager));
9  }
```

Figure 2.13: Adding the custom listeners in AutofetchIntegrator

2.2.1.4 Proxy Handling

Proxies in Hibernate is an essential part of the lazy loading mechanism. As we mentioned earlier, a proxy is generated by Hibernate dynamically to represent the actual object, but without having to gather the information of its fields with queries. Hibernate has switched between different proxy generation frameworks the last decade. The original tool used CGLIB[42], but stagnant development led to the switch to Javassist[43] in the release of Hibernate 3.3. Then in Hibernate 5, Byte Buddy[44] became the default bytecode provider but with support for Javassist. As a consequence, the proxy we implemented in Autofetch 4.3 is an implementation based on the Javassist proxy from Hibernate. In the 5.3 version of Autofetch we still use Javassist, but with plans to migrate to Byte Buddy in the upcoming releases. The two implementations take heavy inspiration from the *JavassistLazyInitializer*. The implementation of the invoke-method can be seen in Figure 2.14, where we intercept calls to the proxy, and before we return the actual value of the underlying object, we add trackers to the proxy in order to collect statistics, as seen in line 18.

```
1  @Override
2  public Object invoke(final Object proxy, final Method thisMethod, final Method
   ↪  proceed, final Object[] args) throws Throwable {
3              result = this.invoke(thisMethod, args, proxy);
4          if (result == INVOKE_IMPLEMENTATION) {
5                  returnValue = thisMethod.invoke(target, args);
6                  if (returnValue == target) {
7                      if (returnValue.getClass().isInstance(proxy)) {
8                          return proxy;
9                      } else {
10                         LOG.narrowingProxy(returnValue.getClass());
11                     }
12                 }
13
14                 return returnValue;
15             } finally {
16                 if (!entityTrackersSet && target instanceof Trackable) {
17                         entityTrackersSet = true;
18                         Trackable entity = (Trackable) target;
19                         entity.addTrackers(entityTracker.getTrackers());
20                         if (entityTracker.isTracking()) {
21                             entity.enableTracking();
22                         } else {
23                             entity.disableTracking();
24                         }
25                     }
26         } else {
27             return result;
28         }
29     } else {
30         if (thisMethod.getName().equals("getHibernateLazyInitializer")) {
31             return this;
32         } else {
33             return proceed.invoke(proxy, args);
34         }
35     }
```

Figure 2.14: The *invoke*-method of the *AutofetchLazyInitializer*-class

2.2.1.5 Serialization of Proxies

There are situations where one might be interested in serializing proxies, such as when you have need to send proxy details from the backend to the frontend and frontend framework does not support Java-classes. In the original tool, one of the disadvantages was that serialization of proxies was not supported. However, in the migrated versions of Autofetch a class *AutofetchSerializableProxy* we have implemented this feature, enabling users to serialize proxies when needed. This results in the tool being applicable in more scenarios. The AutofetchSerializableProxy-class is analogous to the SerializableProxy-class provided natively by Hibernate.

2.2.1.6 Bytecode Enhancement

Hibernate 3.1, the version that Autofetch was originally developed for, used lazy loading as its default fetching strategy. Specifically, it used lazy select fetching for collections and lazy proxy fetching for single value associations. This is optimal for Autofetch since we want to lazy load all associations in order to gather statistics, and then based on these statistics decide what needs to be prefetched. However, the default fetching strategy changed for versions 4.x, and this change is still in effect to this day. The current default strategy depends on the cardinality of the association. The default strategy follows the JPA standard:

- **OneToMany**: *Lazy*
- **ManyToOne**: *Eager*
- **ManyToMany**: *Lazy*
- **OneToOne**: *Eager*

This change implies some problems for the methodology of Autofetch. In order for Autofetch to function the way it is supposed to, it needs to load all associations lazily. The two options for setting the fetch strategy are either to set the associations manually in the mappings file or to use a so called *Bytecode Enhancement*-plugin. The first option is not viable in the case of Hibernate, since the *OneToOne* associations will always be loaded eagerly, no matter what fetch strategy the user sets. This leaves us with the latter, adding Bytecode Enhancement. Bytecode Enhancement is a technique to manipulate classes at build or runtime, making it possible to ”enhance” the functionality, and in this case enabling LazyInitialization of all associations. Fortunately, there is a maven based tool for this in Hibernate which can be added to the pom.xml-file of the project. The tool contains of four main enhancement capabilities, consisting of:

- LazyInitialization
- DirtyTracking
- AssociationManagement
- ExtendedEnhancement

The capability that is of interest in this study is LazyInitializaition, which enables that all associations are lazily instantiated. With this setting set to true, we can expect even OneToOne-associations to be lazy loaded. We added the Bytecode Enhancement tool to the migrated versions of Autofetch and set the capabilities in the tool like in Figure 2.15, where line 10 is where we set the LazyInitialization property.

```
1  <build>
2      <pluginManagement>
3          <plugins>
4              <plugin>
5                  <groupId>org.hibernate.orm.tooling</groupId>
6                  <artifactId>hibernate-enhance-maven-plugin</artifactId>
7                  <version>\${hibernate-orm.version}</version>
8                  <configuration>
9                      <failOnError>true</failOnError>
10                     <enableLazyInitialization>true</enableLazyInitialization>
11                     <enableDirtyTracking>true</enableDirtyTracking>
12                 <enableAssocManagement>true</enableAssocManagement>
13                     <enableExtendedEnhancement>false</enableExtendedEnhancement>
14                 </configuration>
15                 <executions>
16                     <execution>
17                         <id>compile</id>
18                         <phase>compile</phase>
19                         <goals>
20                             <goal>enhance</goal>
21                         </goals>
22                 </execution>
23                     <execution>
24                         <id>test-compile</id>
25                         <phase>test-compile</phase>
26                         <goals>
27                             <goal>enhance</goal>
28                         </goals>
29                     </execution>
30                 </executions>
31             </plugin>
32         </plugins>
33     </pluginManagement>
34 </build>
```

Figure 2.15: The Bytecode Enhancer plugin used to enable LazyInitialization

2.3 Method of Evaluation

The old tool and the original study regarding the Autofetch methodology put it through some benchmarks, including *TORPEDO*[45] and *007*[22], in order to evaluate the gained performance. However, the performance of the tool has only been investigated in one type of scenario, and we aim to extend the spectrum of tests by testing the migrated tool in a real world JavaEE-application, meaning that we evaluate the method in a system developed for the purpose of being used in production. In order to not cause any problems for the customer, we are running the system with the tool integrated against a local backend, and not in production.

Furthermore, since the migrated versions include some changes compared to the original tool, there might be a difference in performance. The question is how well does Aut-

ofetch performs in a real world scenario, and how useful does it prove to be in terms of practicality, ease of use etc. To get a better picture of the performance gained by the tool, we are evaluating the migrated tool on a project of itestra GmbH. With this evaluation, we hope to cover some aspects of performance and usage that the original research did not analyse. The results of the evaluation can then be used in order to answer the questions in table 1.1.

2.3.1 Target Project

The project chosen for executing the evaluation is a project called *TAS*, a sales front end system used to create applications for general insurances for business customers. It is a Java 7 based system using *Java Server Faces* and *Primefaces* together with a Tomcat 7 server. The backend is a rule engine called *VPMS* which is developed by DXC. The application uses Hibernate 4.3 to handle its persistence.

Based on some normal usage patterns that was provided by application's project leader, a small benchmark was created. The benchmark consists of some different operations carried out in the application. Since Autofetch uses statistics from earlier executions to generate the prefetch criteria, the operations within the application are repeated five times. The reason why we opted for five iterations instead of a bigger number in the benchmark is mainly because we could not find a way to programmatically do the actions in the benchmark without spending too much time on rewriting big parts of the code. This would simply take too much time, and since we spent much time on the migration, we opted for few iterations.

The test data used was an already existing test set consisting of 8 created applications, together with 2 manually added applications, together making up a total of 10 applications.

2.3.2 Evaluation Tools

In order to evaluate the tool's tuning capabilities, we need to a method of diagnosing the performed queries for each execution. We are interested in lowering the amount of queries, in order to increase the performance of JavaEE systems especially in situations where network delay is a factor. However, network delay solely does not tell if the performance is good or bad, but the execution time of queries is of interest as well. Therefore, the two aspects we have chosen to focus our tests around are:

- Number of executed queries
- Total execution time of queries

The amount of executed queries and execution time during a session can be monitored natively in Hibernate. In order to get the total amount of executed queries for test, all the printed results of each session can be summed together. The same method can be used for the total execution time. In order to print session statistics in Hibernate, the user has to activate the two following settings in *hibernate.properties*, which can be seen in Figure 2.16, and in the settings-file of the used logging-framework, which can be seen in Figure 2.17.

```
<properties>
<property name="hibernate.generate_statistics" value="true" />
...
</properties>
```

Figure 2.16: Setting for enabling statistics in Hibernate

```
org.hibernate.stat=DEBUG
```

Figure 2.17: Setting the correct logging level for displaying execution time

These two settings will print out all the executed queries together with the execution time of each query and the total amount of executed queries. The data gathered from the log-files generated from the configuration with the tool enabled can then be compared with the configurations without the tool and handtuned respectively. After this comparison, results regarding the performance can be determined.

2.3.2.1 Benchmark

Within TAS, we will run a small test suite from the graphical user interface to simulate normal usage in the application, meaning that the benchmark is performing the most commonly used business cases. These usage patters and use cases were obtained by the project leader of the project. In order to gain insights in how the tool performs in this environment, we need to localize in what actions in the user interface that executes queries. Looking into the source code, we figured out that the following interactions trigger database queries:

- Load an application
- Save an application
- Press save application button
- List existing applications
- Edit a mediator

We need to make sure to design the benchmark with some these points in mind in order to gather enough statistics for the tool to be able to do the tuning. The order of the operations is also relevant, since the tool needs a clear usage pattern in order to do the tuning optimally. Therefore, we have designed the benchmark to execute the interactions mentioned earlier in the following order:

1. List the existing applications
2. Load an application
3. Press save application button

4. Save the application
5. Edit a mediator

We will run the same benchmark five times and observe the differences between iterations. With the data gathered from each iteration of the benchmark, we can then compare the results between different configurations. These are the three different configurations tested:

- Configuration without Autofetch and all associations set to lazy
- Handtuned configuration without Autofetch
- Configuration with Autofetch

All of these configurations will be tested five times. Since we expect the tuning to happen throughout the iterations of the configuration with Autofetch, we have decided to show each iteration of this specific configuration. The other configurations will also be run five times, but instead of showing one bar per iteration, we have decided to calculate the average value of these iterations and represent these configurations with one bar each. This will hopefully make the bar charts more readable.

Furthermore, there are some considerations to be made when applying to the tool to an already existing project. For instance, this work will be executed in a real world JavaEE application, meaning that the application was designed to be used in production, and the design of the application was not originally intended to be used with the Autofetch framework. This could lead to some errors to occur. For example, changing the fetch strategies of the application can cause errors such as objects not being available in cases where they are needed. There are cases where the application is not currently in a session and it tries to load a field of a proxy which has not been instantiated. This can result in exceptions being thrown and potentially lead to application crashes. These problems could potentially be solved with the setting of being able to lazy load outside of transactions or keeping an open session throughout the program duration. These aspects and the consequences of these aspects will be discussed further in the following sections of the report.

Chapter 3
Evaluation

In order for the tool be usable in real world scenarios, which is the main question of this research, it is crucial that the tool simplifies the optimization process in terms of usability, as well as improving the performance of an unoptimized project or perform similar to a handtuned version of a project. Therefore, we will present the results received from the study both from a usability and performance point of view. We will then discuss the obtained results. This will cover mostly the current implementation, and then we will take a broader perspective and discuss the methodology itself.

3.1 Results

As previously mentioned, since we are interested two key aspects in the evaluation, we have divided the evaluation into these two different categories:

- *Integration and Configuration*
- *Performance*

This was done to be able to easier distinguish the two different aspects and keep the evaluation of these two aspects separated. We will start with the integration since that is the first aspect that the user will have to handle. After that, we will cover the performance of the tool in this specific case.

3.1.1 Integration and Configuration

Integrating Autofetch into a project is a process that can be done simply by adding the dependency to the tool in Maven, or manually add the generated JAR-file to the project and the rest should be automatic. If this was the case, then the code snippet from Figure 3.1 could be added to the pom.xml-file to integrate the tool.

```
<dependencies>
    <dependency>
        <groupId>org.autofetch</groupId>
        <artifactId>autofetch</artifactId>
        <version>0.1-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Figure 3.1: Enabling Autofetch in a Maven based application

However, since TAS is not using Maven, the integration of Autofetch into TAS was more difficult than expected. This meant that a JAR-file containing the Autofetch-tool had to be generated through Maven, and then imported manually to the target project, instead of simply adding the Autofetch dependency to the pom.xml-file. Because the target project did not use Maven, the dependencies were initially inconsistent and created *ClassNotFoundExceptions* and *NoClassDefinitionExceptions* was thrown. Handling these inconsistencies in the classpaths manually is a cumbersome task that can take vast amount of time, depending on the project structure complexity and size. In this case, the problem was eventually found and could be solved, however it took two days of work for one inexperienced developer to figure out. If the target project had been using Maven, this whole situation could be avoided and the integration process could have taken a few minutes instead of days of work. Transforming the target project to Maven was also tried, but since we had no earlier experience with Maven, we did not succeed in doing this.

Once the integration was done, we could use the tool without having to write a single line of code or enable the tool in settings-files, and since we saw no reason to change the default settings on how the fetching should be done, such as *Fetch-depth* and *Fetch probability threshold*, we spent no time configuring the settings.

Since the application was handtuned by default, we decided to do remove any explicit fetch strategies from the entity mapping files in order for Autofetch to function as it should. The fetch strategies can be left as they are, however it can affect performance since the tool expects the fetch strategy to be lazy for all associations. The problem with having to change the fetch strategies is that in many cases, the application has been specifically developed for certain fetch strategies for every association, and when that changes, the application does not function correctly. In the case of TAS, this was also a problem. Therefore, we had to enable the widely debated *hibernate.enable_lazy_load_no_trans*-feature in Hibernate to let the application load lazy associations freely. This feature can be enabled by adding the snippet from Figure 3.2 to the persistence.xml-file or the hibernate.cfg.xml-file. We will discuss this feature and its consequences more in detail in the *Discussion* section.

```
<propertyname="hibernate.enable_lazy_load_no_trans" value="true"/>
```

Figure 3.2: Enabling lazy loading outside of transaction scope in Hibernate

3.1.2 Performance

In this section, the performance of the application when applying the Autofetch-tool will be presented. Initially, the benchmark of the application was run without any fetch strategy directives five consecutive times, in order to inspect how many queries that were being executed and how long execution time each query used. This represents the performance of a non-tuned version of the application.

The values obtained in the benchmark can be seen in table 3.1. In the rows we can observe the type of interaction in the application. In the columns we see the different configurations, which in our case are the lazy loaded version, the handtuned version, and the different iterations of Autofetch. We display every iteration of Autofetch because we want to see how the performance changes for every iteration. As mentioned before, the other configurations are also executed five times, however they execute the same number of queries in every iteration.

Interaction Type/Configuration	**Lazy**	**Handtuned**	**AF1**	**AF2**	**AF3**	**AF4**	**AF5**
List all applications:	65	25	65	58	56	56	56
Load an application:	30	16	30	30	30	30	30
Press save application button:	1	1	1	1	1	1	1
Save application:	9	9	9	9	8	8	8
Save Vermittler:	10	5	10	7	7	7	7

Table 3.1: The number of executed queries for each configuration including the five iterations of the Autofetch configuration

In order to get a visual representation of the results, we present the results in the form of bar charts in Figures 3.3, 3.4, 3.5, 3.6, 3.7, 3.8.

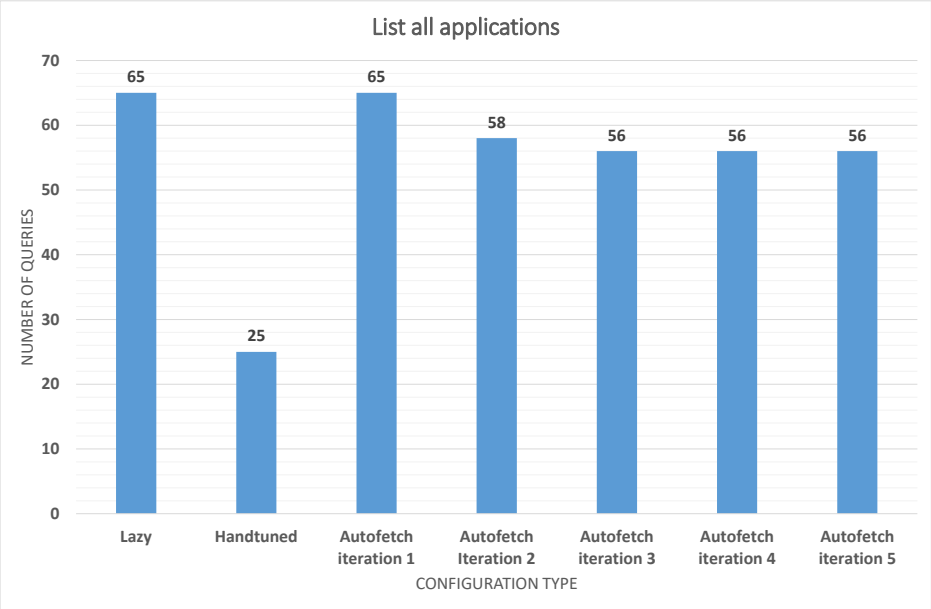

Figure 3.3: Graph displaying the number of executed queries for the List all applications-action in TAS

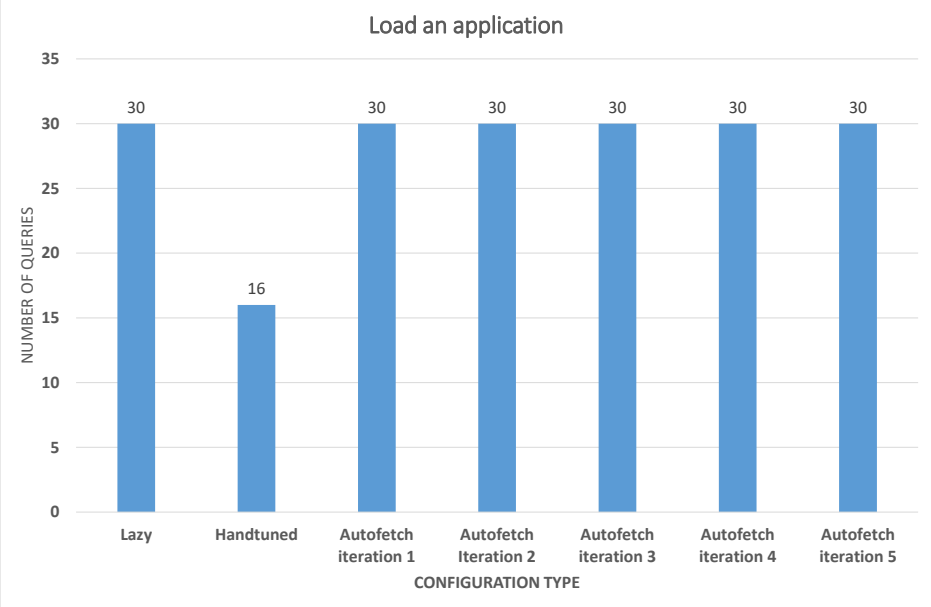

Figure 3.4: Graph displaying the number of executed queries for the Load application-action in TAS

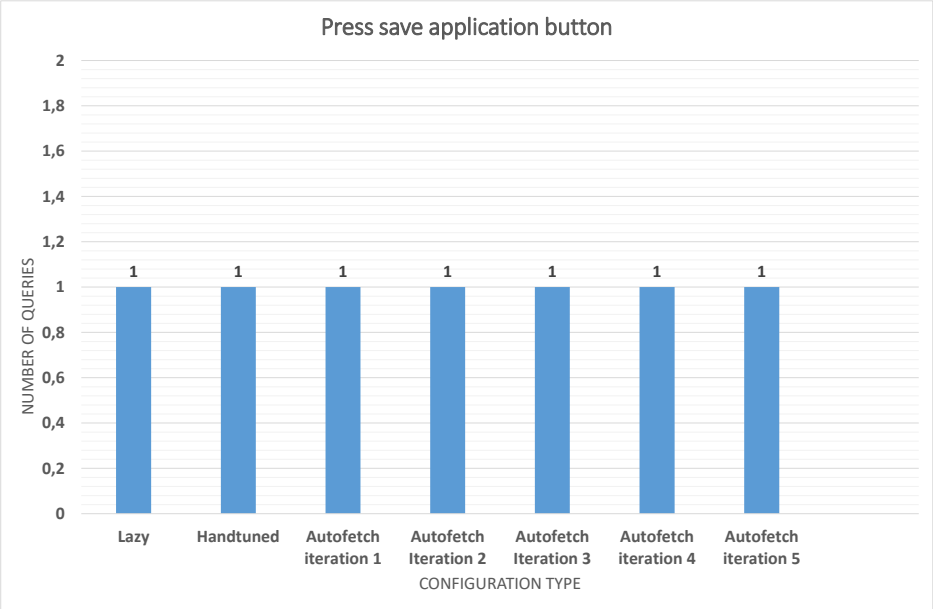

Figure 3.5: Graph displaying the number of executed queries for the Save application-action in TAS

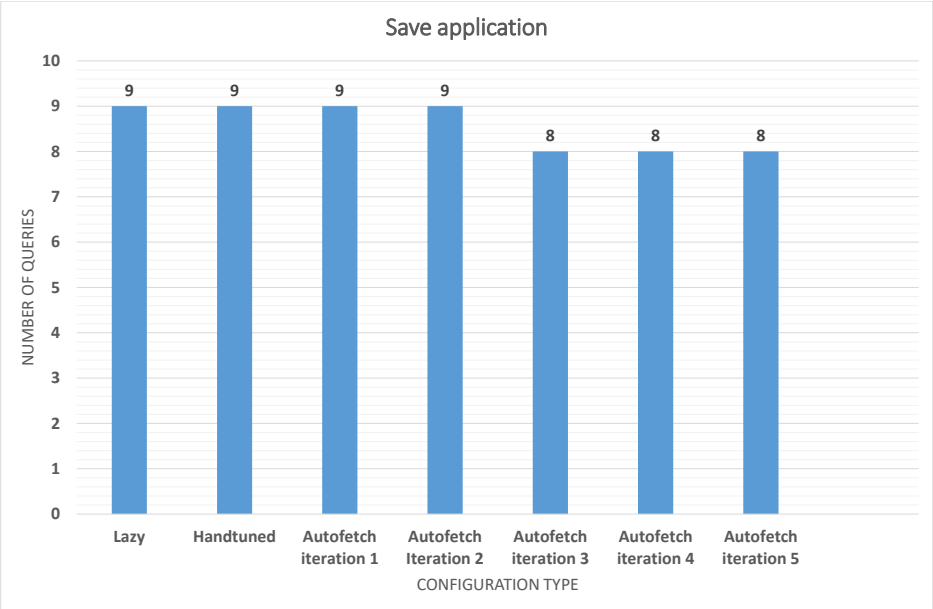

Figure 3.6: Graph displaying the number of executed queries for the Save application-action in TAS

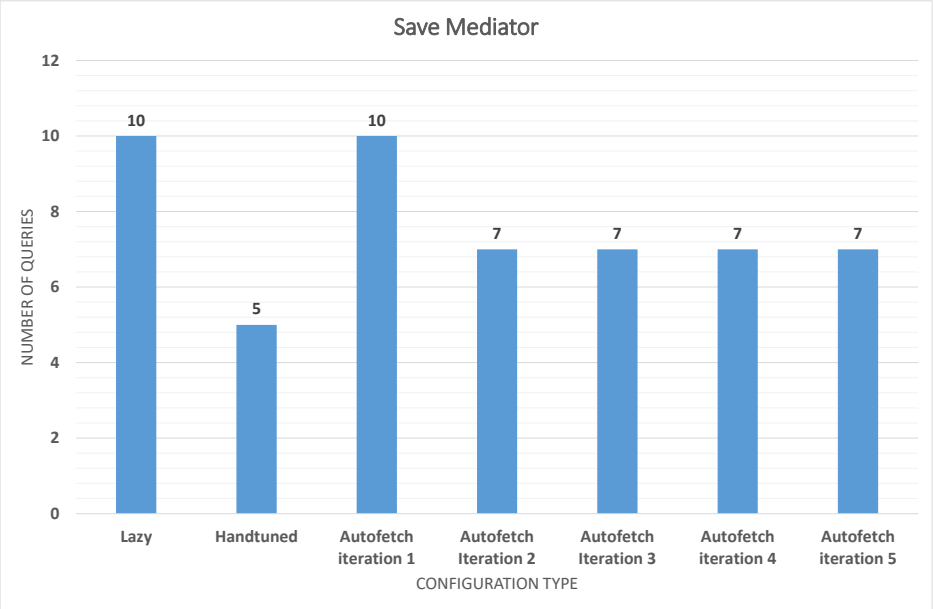

Figure 3.7: Graph displaying the number of executed queries for the Save mediator-action in TAS

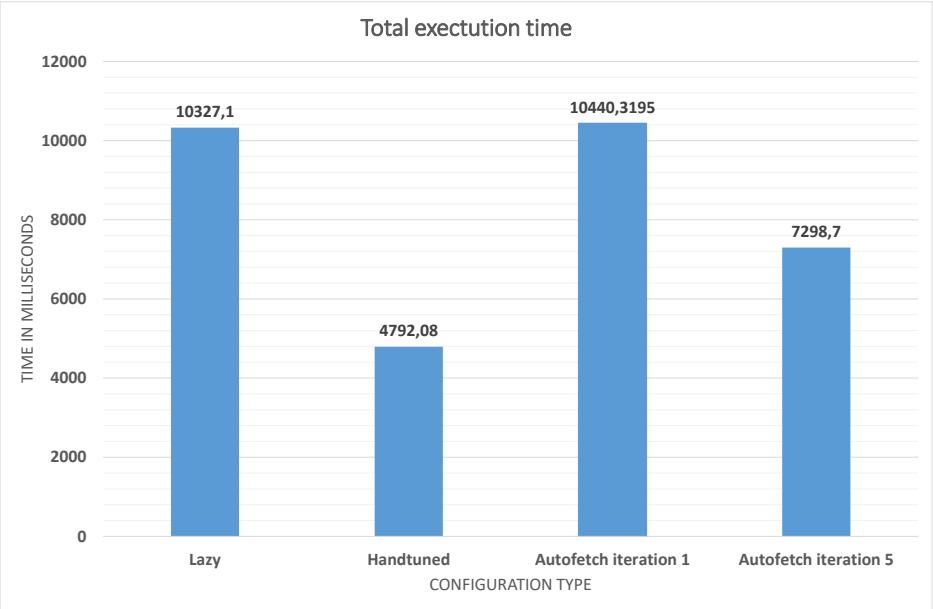

Figure 3.8: The total execution time of each configuration

The obtained values for the tests prove that indeed the tuning takes place. However, the tuning is not of the same magnitude as the authors of the original Autofetch article claim in their study, which can be a result of various factors, such as the implementation being slightly different and that the test application is different. We will discuss this matter further in the discussion section. We can observe that the tuning only take place in the *List all applications* and *Save application* scenarios in Figure 3.3 and Figure 3.6 respectively. In the cases of *Load an application*, *Press save application button* and *Save mediator*, no tuning is being done over the five iterations. Apparently, our tests only cover loading of some associations and not all of them. Throughout the entire benchmark, the Autofetch configuration executes 11.3% fewer queries than the lazy configuration. The manually configured version performs considerably better still, with 51.3% fewer queries executed in comparison to the lazy loaded configuration. Furthermore, the total execution time of each configuration can be seen in Figure 3.8 and shows manually tuned configuration is the faster overall option in terms of execution time, followed by the Autofetch configuration after five iterations. The total execution time is correlated with the amount of queries being executed.

3.2 Discussion

Autofetch for Hibernate is still a work in progress, and with the obtained test results it is clear to say that the tool does not perform as well in our use case as the handtuned version of the TAS application. With 11.3% fewer queries executed compared to the lazy loaded variant, it did offer a slight improvement in terms of performance. The improvements are small however and do not match the improvements discussed in the original Autofetch-article, where the authors claimed to have increased the performance of various applications with up 99.8% in terms of eliminated queries.

The exact reason for why this version of the tool does not match the claimed performance of the old tool is still not quite certain, but comes down to a few different aspects. We will go through these here.

First of all, Hibernate as a framework has changed considerably since version *3.1* compared to *4.3.10* that we tested in this research. This might cause that the implementation might need reconsideration.

Moreover, the test project did not contain a complex association mapping and therefore could not use Autofetch to its full capacity. The way that TAS uses Hibernate is simple in terms of persistence and the dependencies between different entities are minimal. Therefore, a new test containing more test cases could lead to completely different results.

Additionally, the test benchmark is based on what we know about the project, and since they have no deeper knowledge about the project, there could potentially be improvements to be made to the benchmark that we used to test the tool. An additional aspect that might have affected the obtained results is the fact that we used a small dataset. This dataset is realistic for some users of the TAS application. However, there are users that have up to a thousand of applications, compared to the ten tested applications we used in our benchmark. In a more in depth evaluation of the tool, datasets of different sizes should be used. This is something we wanted to do, however due to time restrictions and the process of migrating the tool taking longer than expected, this could not be done. With

more extensive datasets we could have performed the benchmark in an environment more similar to a production environment. With think that with more data for the tool to analyse, the tool would have more statistics to base the tuning on, and therefore it could tune more precisely.

Furthermore, in order to get the application to run with Autofetch, we needed to enable the setting *hibernate.enable-lazy-load-no-trans*, which lets Hibernate lazy load freely, without having an active session. Not only does this open up sessions without the user knowing about it and therefore negatively impacting performance of the application, but this setting also has a famous reputation of inducing the N + 1 problem when activated[47]. Autofetch should handle the issue regarding the N + 1 problem with this setting, however the heavy increase of created sessions in the application will affect the general performance of the application. This constitutes yet another source of error.

Additionally, similar to the topic discussed in the first point in this section, the changes to Hibernate does not only cover what has been done to the framework itself, but the underlying framework of JPA has also changed drastically. In earlier versions of Hibernate, JPA has changed the standard fetching strategies for some association types. For example, ManyToOne and OneToOne associations are fetched eagerly by default, and for the parent side OneToOne-mapping, the fetch strategy can not be changed unless a Bytecode Enhancer is used. As already mentioned, the 4.3.10 version of Hibernate does not have a working Bytecode Enhancer. These changes make the tested version of the tool to potentially function differently from the original tool.

The lack of documentation about the original tool also adds a layer of uncertainty to how the tool is supposed to function, and in what scenarios it optimizes optimally. In our case, we had to thoroughly analyse the source code in order to make assumptions about what settings to use in Hibernate in order to achieve the biggest optimizations. These assumptions might not be entirely correct, and could therefore be a source of error.

In terms of usability, the tool has been simplified and integrates itself once the JAR is in the buildpath or when the snippet has been added to the pom.xml-file. Using a tool that will handle the prefetching by just having it added to the project is a promising feature and with future enhancement of the tools tuning capabilities the methodology could truly bring big improvements to the Software Engineering aspects of JavaEE-projects.

Chapter 4
Conclusions

In this chapter, we will present the findings of the report including shortcomings of the current tool and the study as a whole, possible improvements, lessons learned from the research together with a section about relevant topics of research in this field.

4.1 Implementation

4.1.1 Integration and Configuration

Compared to the first version of Autofetch, the way of integration has changed drastically. Executing an Ant-script, adding JAR-files manually to the classpath of the project can be a daunting task. With the new integration mechanism introduced in the migrated version of Autofetch, the procedure has been simplified and less error prone; add the Autofetch dependency to the *pom.xml*-file, and the integration of Autofetch into Hibernate is automatic. Even though the improved usability due to automatic integration when using Maven could not be used in this specific case, we would argue that this simplified the use of the tool significantly and that in order to make the tool user-friendly, an integration mechanism like the one we developed will make the experience with the tool better.

Regarding the configuration, there was not many changes done from the old tool. For example, setting the depth of the object graph and turning on and off tuning, remained unchanged. In order to make the tool easier to integrate, a future improvement could be that Hibernate includes Autofetch by default.

4.1.2 Performance

Autofetch lowered the amount of queries executed in comparison to a completely lazy loaded configuration with 11.3%. However, the handtuned version performed better and

lowered the amount of queries executed with 51.3%. The general rule of thumb regarding the total execution time of the different configurations seemed to be proportional to the amount of queries being executed, resulting in an average execution time for each query in each of the configurations to be similar.

4.2 Limitations

When migrating the tool, the goal was to create minimal viable product with just the core functionality of automatically adding prefetch directives. This was done due to time-constraints, and if the Hibernate community wants to continue to develop the tool it is easy to do so due to the open source nature of the project. Due to the time-constraints, there are also some unhandled faults to the implementations of the tool.

The most apparent issue is that Hibernate uses the default JPA fetch strategies for association, meaning that there are certain fetch strategies for association types that are not overridable. The result is that these associations always will be eagerly fetched, no matter what the developer defines for the association. This means that it has to be done some other way, with for example the use of the *Bytecode enhancement*-plugin to enforce lazy loading for all associations. However, in the *4.3*-version of Hibernate, the bytecode enhancement maven plugin is not supported. A developer of the tool cover this in an article where he explains that the functionality of the tool is not correct until the *5.x*-version of Hibernate[41]. This is a technique that is not documented well and therefore we had problems implementing it correctly. With some continued development of the tool, this could be resolved either by just supporting Hibernate *5.0* and onwards, or by submitting a pull request with a fix to the repository of the Maven Bytecode enhancer tool. In terms of practicality, the easiest way to ensure a working Autofetch-tool would be to simply limit the compatibility to newer versions, since there is no continued development of the older versions of the framework.

It should be noted however, that the fact that this implementation heavily relies on the old *3.1*-version of Hibernate, which was an implementation that did not have official support from the Hibernate team. This resulted in that the developers could not make the necessary adjustments to the Hibernate core in order to implement tool in an optimal way. This still stands today, and during the development and migration of the tool we constantly ran into situations where we needed to make a change for methods of the core, but since we did not have control of these parts of the code, we could not carry out the necessary changes, and we had to opt for a change that was not optimal. In some cases, the new versions of Hibernate even made certain methods private, disabling us from extending the necessary functionality. An example of this can be seen in Figure 4.1, where we see the change to the method in which the foundation of the added loading mechanism behaviour is placed, suddenly its access modifier changed to private, hence making overriding this method impossible. The before the change version is shown on line 2-4 and the version after the change is shown on lines 5-7.

```
1     @Override
2   - protected Object loadFromDatasource(LoadEvent event,
3   EntityPersister entityPersister, EntityKey entityKey,
4   LoadType loadType) throws HibernateException {
5   + private Object loadFromDatasource(LoadEvent event,
6   EntityPersister entityPersister, EntityKey entityKey,
7   LoadType loadType) throws HibernateException {
8
9       String classname = entityPersister.getEntityName();
10      if (log.isDebugEnabled()) {
11          log.debug("Entity id: " + event.getEntityId());
12      }
13      // Autofetch logic
14          result = super.loadFromDatasource(event, entityPersister, entityKey,
            ↪  loadType);
15      }
16      extentManager.markAsRoot(result, classname);
17      return result;
18  }
```

Figure 4.1: The method *loadFromDatasource* being changed from *protected* to *private*

This leads us to the conclusion that the only viable option for continued development would be if the Hibernate developers themselves integrated a custom implementation of Autofetch, or at least that the developers of the tool had close contact with the Hibernate developers in order to discuss necessary changes. We will discuss this more in detail in the section *Future Improvements*.

Regarding the evaluation of the tool and the target project TAS, there were also some shortcomings. For example, the dataset is basically too small to draw any definitive conclusions whether Autofetch is a good match for JavaEE applications or not. We used a data set of ten so called applications, which are the entities of TAS. While this covers some of the normal use cases of this application, we should have covered more use cases by altering the benchmark and increasing the amount of entities from 10 to a larger number which covers all the use cases of the application. It would also be interesting to see Autofetch in other applications other than TAS.

4.3 Lessons Learned

In this attempt to evaluate the methodology of Autofetch there were a few shortcomings that made it difficult to draw any definitive conclusions. Initially, the idea was to have the research more focused on evaluating Autofetch in a real world scenario, rather than the migration of the old tool. In addition, the idea was to apply the tool on different projects in **itestra GmbH** in order to have a bigger data pool to draw conclusion from. However, since the migration of the tool did not go as seamless as expected, the focus had to be slightly adjusted. Instead, this work in the end focuses more on the migration and only analyses the test results from one project instead of the three projects, which was the original intention. These three test systems would be of different sizes in terms of the usage of Hibernate, and potentially systems of different domains, such as *insurance, banking* and *manufacturing*. If the projects used a similar Hibernate-version, it would be easier to apply the tool. However, in reality the version differs heavily between these projects and

therefore it would be needed to make individual migrations for each version, making it too cumbersome and time-consuming for one person to develop.

There were also other factors that made the evaluation more difficult than anticipated. For example, the test system chosen for evaluation proved to be difficult to work with. Not only is the system using an older version (4.3), making it difficult to find proper support and help in the forum for this specific version, but also since it is based on user interaction, which makes it difficult to do extensive testing without spending massive amounts of time. If we could have done the benchmark in the form of unit tests, we could have done far more iterations of the tests. We tried implementing unit tests, however we found it difficult to do similar actions to what a user does using unit tests, since we do not have prior experience with TAS.

Another challenge with the target project is that it does not use Maven, meaning that the integration with Autofetch is not as seamless as it could have been. Without Maven, JARs have to be generated and then manually added to the project, which seems easy in theory but can be burdensome in cases with complex project structure, such as Web-applications. Changing the build path was a very error-prone process which ended up taking a lot of time from the actual research. Choosing a project that used Maven could save countless hours which could be invested in developing the tool and analysing the results. If the research were to be repeated, a more suitable target system should be chosen, meaning that the system should be Maven based and using a version 5.x of Hibernate. This would ensure easier integration with Autofetch, but also make it easier to attract more contributors to the open source project, since most people do not want to develop for a legacy version of Hibernate.

For future reference, if similar research was initiated today, we would have chosen to focus primarily on one topic. In the case of this research, the main objective was to evaluate the method. However, since the old tool was in a legacy state and had to go through the migration, we could not do a thorough evaluation due to time concerns. In hindsight, focusing solely on the migration while highlighting some improvements of the new implementation could have been a more realistic scope.

4.4 Future Improvements

During the research and development of the tool, there were a few improvements that became apparent. In this segment of the report, we will list these potential improvements. The current versions of Autofetch only contain the most core aspects of the methodology. Some functionality was meant to be in the tool, but was omitted due to time constraints.

- Saving statistics between executions
- Feature enabling overriding of fetch strategy
- Add support for more data structures than Set, List, Bag and IdBag

In the current implementation of the tool, we only gather statistics from the current execution of the program. This might not be a problem in production for *JavaEE*-systems, since the execution time for these type of systems lasts for a long duration of time. However, for smaller type of systems this might be a problem, and especially if execution stops

frequently. This would mean that the gathered statistics would be lost, and for the next execution all the statistics would have to be gathered again, leading to lowered performance initially.

In the Ebean implementation, this has been solved by saving the context information together with the statistics in a xml-file, allowing the statistics of earlier executions to be used in prefetching decisions by Autofetch in future executions. This can be seen in Figure 4.2, where the extent information is stored in a xml-file that can be used in later executions of the program. A similar extension to the Hibernate-tool would increase the usability and is something that should be prioritized in future development.

```
1  <autotune xmlns="http://ebean-orm.github.io/xml/ns/autotune">
2      <origin key="wpbnw.BQejAr.Bzpaqe" beanType="org.example.domain.Order"
3          detail="select (orderDate,shipDate,status)  fetch details (id,orderQty)"
4          original="select ">
5      <callStack>org.example.domain.finder.OrderFinder.byStatus(OrderFinder.java:38)
6          org.example.domain.finder.OrderFinderTest.test(OrderFinderTest.java:20)
7          sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
8          sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
           ↪ java:62)
9          sun.reflect.DelegatingMethodAccessorImpl.
           ↪ invoke(DelegatingMethodAccessorImpl.java:43)
10     </callStack>
11     </origin>
12 </autotune>
```

Figure 4.2: A snippet from the file that stores the extent information in Ebean

In general, the methodology would greatly benefit from getting official support and integration from the Hibernate software group. Today, due to integration difficulties with the Hibernate core, the possible benefits of the tool are more limited in terms of functionality. If the Hibernate group was to continue the development of the Autofetch-tool and make changes to the Hibernate library so that the implementation could be more seamless, we could see a lot of potential use cases and users. The Ebean-implementation is a good example of that, where it is natively included in the framework and the activation of Autofetch is just a flag in the *ebean-autotune.xml*-file.

Lastly, in the current version of the tool, not all the commonly used data structures have an Autofetch counterpart. This means that applications using certain data structures can not use the tool. This is a change that would be easy implement since the collections wrappers does not hardly contain any logic but simply does what the original data structure does but tracks the usage. The additional wrappers would be analogous to the other custom collection types in Autofetch. This was left out due to time restrictions.

4.5 Future Work

This research barely scratches the surface on the topic of Automatic prefetching in persistent systems. First of all, this work made an attempt to migrate the method to the latest release of the most popular ORM-mapper on the market, Hibernate. Then we proceeded

to test it on a small JavaEE-web application, however not in production. In future evaluations, we hope that the problems of migrating the tool to the latest version should be eliminated and that the focus can be how well Autofetch performs in an enterprise type software of different sizes and in a production environment. This would constitute a more valid display of the methodology's capabilities.

There are other interesting concepts similar to this that could be investigated further. For example, some developers do not like the idea of automatic prefetching and would rather do it themselves. For this type of developers, similar techniques described in the original Autofetch report could be used to advise developers about what fetch strategy that is optimal in certain places in the source code. With this approach, developers could have full control while still receiving useful tips about the optimal prefetch strategy in different scenarios. This would complement the functionality of the methodology and make it more versatile, whether the user wants it fully automatised or just wants some useful input from the tool. To make it even more interesting, the idea could be brought into development environments, such as *Eclipse* and *IntelliJ*. This way the developer could get useful statistical information about the usage of associations directly in the development environment. Using this approach, the user could achieve high performance using the statistics, or even applying prefetching recommendations for certain associations, while avoiding the problems of having the fetch strategies completely auto generated, such as caching problems and not having an active session. We think that this approach would be a viable option to auto generated prefetching.

Furthermore, there are aspects of this technique that should be considered. For example, we do not look into how much overhead the tool adds in terms of memory and CPU usage. In order for the methodology to be worth considering for production environments, the tool must run efficiently. Once the methodology has an implementation that has full support from the persistence framework provider, it would be interesting to perform more extensive research on different projects in their production environments. This would lead to results that would be more relevant in terms of applicability than the attempt carried out in this research.

4.6 Summary

In the first chapter we defined some questions that we wanted to answer in this work, so in this brief summary we will present the conclusions based on the initial problem formulations formulated in table 1.1. As described in this report, Autofetch has been migrated and can be applied to a project using Hibernate version 4.3.10Final. We have shown that the tool can easily be integrated and configured to any project, preferably with Maven. The tool does reduce the amount of loaded queries compared to a configuration loaded completely lazy, and therefore improving performance. Reducing the amount of executed queries is generally a good optimizing measure, even more so in systems where the round trips to the database is affected by network delay. Therefore, the usage of Autofetch in JavaEE environments could increase performance for systems.

However, in the case of the test project used in this work, better performance could be achieved by manual tuning of the association mappings. The results show that the lazy loaded configuration generates 115 queries, and the Autofetch configuration executes 102

queries, resulting in a 11.3% less queries being executed. However, the hand tuned configuration generates 56 queries, which is a decrease of 51.3% executed queries compared to the lazy loaded configuration, meaning that in this specific case it is better to opt for a hand tuned configuration rather than to let Autofetch handle the prefetching.

Bibliography

[1] Dearle, A., Kirby, G. NC. and Morrison, R. ”Orthogonal persistence revisited.” International Conference on Object Databases. Springer, Berlin, Heidelberg, 2009.

[2] Jordan, M. and M.P. Atkinson. ”Orthogonal persistence for Java—A mid-term report.” Morrison .[161] (1999): 335-352.

[3] Khan, M., and M. N. A. Khan. ”Exploring query optimization techniques in relational databases.” International Journal of Database Theory & Application 6.3 (2013): 11-20.

[4] Ambler, Scott W. ”Mapping objects to relational databases: What you need to know and why.” Ronin International (2000).

[5] Ireland, C., and D. Bowers. ”Exposing the myth: object-relational impedance mismatch is a wicked problem.” DBKDA 2015, The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications. IARIA XPS Press, 2015.

[6] Ireland, C., Bowers, D., Newton, M., and Waugh, K. ”A classification of object-relational impedance mismatch.” Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA’09. First International Conference on. IEEE, 2009.

[7] Chen, T. H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. ”Detecting performance anti-patterns for applications developed using object-relational mapping.” Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.

[8] Chen, T., Shang, W., Yang, J., Hassan, A. E. and M. W. Godfrey. ”An empirical study on the practice of maintaining object-relational mapping code in java systems.” Proceedings of the 13th International Conference on Mining Software Repositories. ACM, 2016.

[9] Van Zyl, P., Kourie, D. G., and Andrew B. ”Comparing the performance of object databases and ORM tools.” Proceedings of the 2006 annual research conference of the

South African institute of computer scientists and information technologists on IT research in developing countries. South African Institute for Computer Scientists and Information Technologists, 2006.

[10] Bernstein, P. A., S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In Proceedings of the 25th VLDB Conference, 1999

[11] Ghandeharizadeh, S., and A. Mutha. "An evaluation of the hibernate object-relational mapping for processing interactive social networking actions." Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services. ACM, 2014.

[12] ODB official website, https://www.codesynthesis.com/products/odb/

[13] NHibernate official website, http://nhibernate.info/

[14] Ebean github page, http://ebean-orm.github.io/

[15] DB-Engines November 2018 database popularity rating, https://db-engines.com/en/ranking_categories

[16] EclipseLink official website, http://www.eclipse.org/eclipselink/

[17] Hibernate official website, http://hibernate.org/orm/

[18] JPA IBM information page, https://www.ibm.com/support/knowledgecenter/en/SSAW57_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/cwlp_jpa.html

[19] Hibernate 5.3 Documentation, http://hibernate.org/orm/documentation/5.3/

[20] Hibernate optimization guide, https://www.thoughts-on-java.org/tips-to-boost-your-hibernate-performance/

[21] Introduction to JPA fetchtypes, https://www.thoughts-on-java.org/entity-mappings-introduction-jpa-fetchtypes/

[22] Carey, M. J., DeWitt, D. J., & Naughton, J. F. (1993). The 007 benchmark (Vol. 22, No. 2, pp. 12-21). ACM.

[23] The hibernate.enable_lazy_load_no_trans Anti-Pattern, http://hibernate.org/orm/documentation/5.3/

[24] Joshi, A. and S. Kukreti. "Object Relational Mapping in Comparison to Traditional Data Access Techniques." International Journal of Scientific Engineering Research 5.6 (2014).

[25] Ibrahim, A, and W. R. Cook. "Automatic prefetching by traversal profiling in object persistence architectures." European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 2006.

[26] Thoughts on Java, JPA 2.1 Entity Graph : Part 1- Named entity graphs

[27] Ibrahim, A. H. Practical transparent persistence. The University of Texas at Austin, 2009.

[28] Patterson, D. A. ”Latency lags bandwith.” Communications of the ACM 47.10 (2004): 71-75.

[29] Neward, T. ”The Vietnam of Computer Science”, http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

[30] Lascano, J. E. ”JPA implementations versus pure JDBC.” http://www.espe.edu.ec/portal/files/sitiocongreso/congreso/c_computacion/PaperJPAversusJDBC_edisonlascano.pdf 2008.

[31] Hibernate Migration Guides, https://github.com/hibernate/hibernate-orm/wiki/Migration-Guides

[32] Ward, A., and D. Deugo. ”Performance of Expressions in Java 8.” Proceedings of the International Conference on Software Engineering Research and Practice (SERP). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

[33] Hibernate 4.3 Developer Guide, http://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html_single/

[34] Moss, J. E. B., and A. L. Hosking. ”Approaches to adding persistence to Java.” Proceedings of the First International Workshop on Persistence and Java. Sun Microsystems, 1996.

[35] Alvarez-Eraso, D. A., and F. Arango-Isaza. ”Hibernate and spring-An analysis of maintainability against performance.” Revista Facultad de Ingeniería Universidad de Antioquia 80 (2016): 97-108.

[36] Ramachandra, K., and S. Sudarshan. ”Holistic optimization by prefetching query results.” Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012.

[37] Aoki, Y. and S. Chiba. ”Performance improvement for persistent systems by AOP.” Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies. ACM, 2007.

[38] Garbatov, S, Cachopo, J., and J. Pereira. ”Data access pattern analysis based on bayesian updating.” Proc. of the Simpósio de Informática (INFORUM)(Lisbon, Portugal (2009).

[39] Garbatov, S. ”Data access pattern analysis and prediction for object-oriented applications.” INFOCOMP 10.4 (2011): 1-14.

[40] Touma, R., Queralt, A., Cortes, T. ”Predicting Access to Persistent Objects Through Static Code Analysis.” Advances in Databases and Information Systems. Springer, Cham, 2017.

[41] Mihalcea, V. (2018). How does the bytecode enhancement dirty checking mechanism work in Hibernate 4.3 - Vlad Mihalcea. https://vladmihalcea.com/hibernate-4-bytecode-enhancement/ [Accessed 30 Jun. 2018].

[42] CGLIB github webpage, `https://github.com/cglib/cglib/wiki`

[43] Javassist official webpage, `http://www.javassist.org/`

[44] Byte Buddy official webpage, `https://bytebuddy.net/#/`

[45] Martin, B. E. (2005, April). Uncovering database access optimizations in the middle tier with TORPEDO. In Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on (pp. 916-926). IEEE.

[46] Official website of VPMS, `http://www.dxc.technology/life_and_wealth/offerings/22990/58122-vp_ms`

[47] Hibernate blog post about the setting *enable lazy load no trans*, `https://vladmihalcea.com/the-hibernate-enable_lazy_load_no_trans-anti-pattern/`

EXAMENSARBETE Migration & Evaluation of Automatic Query Hint Generation in Persistent Systems
STUDENT Erik Jonasson
HANDLEDARE Per Andersson (LTH), Arnaud Fietzke (itestra GmbH)
EXAMINATOR Flavius Gruian (LTH)

Automating Optimization Procedure for Relational Databases

POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Jonasson**

Optimizing how and when data should be fetched from the database can be a difficult and frustrating task. The bigger the application, the more difficult it becomes. Luckily, there is a way to automate this process.

In today's software development, it is common to use so called *Object-relational-mappers*. What this software does is that it creates an extra layer between the database and the application code. The software will then take care of all database manipulation, translating between the two different worlds of object oriented program code and the relational database.

This type of software can be difficult to configure accordingly. The options are many, and the not so experienced developer can run into many misconfigurations that will cause the system performance to drop drastically.

The difficulty comes when the user has to define the *fetch strategy* for associations of entities in the database. This leaves a lot of room for customization, but also for errors. The common fetch strategies are:

- **Eager loading**: When loading this object, *eagerly* annotated associations to this object will be loaded immidiatelly.
- **Lazy loading**: When loading this object, *lazily* annotated associations will be loaded only when used in the code.

Lazy loading can seem like a better option, since the object will only get loaded when it is being accessed in the code. However, this will create a query to the database for each time when a new object or field is accessed that is not yet in memory. If the database in on a server with latency for example, this could lead to heavily decreased performance. Moreover, manually handling these settings is not optimal from a Software Engineering standpoint.

Due to these problems, we migrated and evaluated an existing tool that automates the prefetching based on statistics of the objects in the database. Since this tool was created for a legacy version of *Hibernate*, we decided to migrate the old tool to the newest version of Hibernate and then evaluate the method on a project of the software consulting company **itestra GmbH**.

The migration introduced new integration mechanisms to make it easier to use the tool. The tool decreases the amount of executed queries by 11% in our tests compared to the version loaded lazily. However, the handtuned version still performs considerably better. On the other hand, with the *Software engineering* benefits that the tool offers, the methodology has potential to become a viable option for anyone who wants to improve performance of the ORM-tool without wasting valuable development time.