

# Master Thesis Software Engineering

Monday, January 11, 2010

## **Theory and experimental evaluation of object-relational mapping optimization techniques**

How to ORM and how not to ORM

by

**Jeroen Bach**

Supervisors: Hans Dekkers & Jurgen Vinju

Company supervisors: Ingrid van Zaanen & Co Kooijman

Publication status: Openbaar

Version: 1.0



## Contents

<b>1</b>	<b>ABSTRACT .....</b>	<b>4</b>
<b>2</b>	<b>INTRODUCTION.....</b>	<b>5</b>
<b>3</b>	<b>RELATED WORK .....</b>	<b>6</b>
3.1	PATTERNS AND TECHNIQUES .....	6
3.2	PERFORMANCE MEASUREMENTS.....	6
3.3	THE OBJECT-RELATIONAL IMPEDANCE MISMATCH.....	7
<b>4</b>	<b>INTRODUCTION TO HIBERNATE.....</b>	<b>8</b>
4.1	RELATION QUERYING CONFIGURATION PRINCIPLES .....	9
4.2	MAPPING CONFIGURATION .....	11
4.3	APPLYING THE TYPE OF MAPPING CONFIGURATION .....	12
4.4	APPLYING THE FETCHING STRATEGY .....	14
<b>5</b>	<b>THEORY OF HIBERNATES QUERY BEHAVIOUR.....</b>	<b>15</b>
5.1	A NOTATION FOR QUERY BEHAVIOUR .....	15
5.2	BASIC BEHAVIOUR .....	16
5.3	IMPORTANT FACTORS.....	19
5.3.1	<i>Bidirectional relationship .....</i>	<i>19</i>
5.3.2	<i>Lazy configuration.....</i>	<i>19</i>
5.3.3	<i>Join fetching .....</i>	<i>20</i>
5.3.4	<i>Recursive relationship .....</i>	<i>21</i>
5.3.5	<i>Concatenating relationships.....</i>	<i>21</i>
<b>6</b>	<b>RESEARCH METHOD.....</b>	<b>22</b>
6.1	MEASURING THE PERFORMANCE.....	22
6.1.1	<i>Environment set up .....</i>	<i>22</i>
6.1.2	<i>Test set up.....</i>	<i>23</i>
6.1.3	<i>Measurement set up .....</i>	<i>24</i>
6.2	PREVENTING BAD PERFORMANCE ONE TO MANY RELATIONSHIP IN ORACLE.....	24
6.3	FACTORS INFLUENCING THE PERFORMANCE.....	25
6.3.1	<i>Object graph.....</i>	<i>26</i>
6.3.2	<i>Mapping configuration .....</i>	<i>26</i>
6.3.3	<i>Actions on object graph .....</i>	<i>27</i>
6.3.4	<i>Storage.....</i>	<i>27</i>
6.4	STABILISATION .....	27
<b>7</b>	<b>PERFORMANCE MEASUREMENTS.....</b>	<b>28</b>
7.1	HYPOTHESES AND RESULTS.....	29
7.1.1	<i>Testing the type of queries .....</i>	<i>29</i>
7.1.2	<i>Testing the amount of queries .....</i>	<i>33</i>
7.1.3	<i>Testing all factors.....</i>	<i>34</i>
7.2	THREATS TO VALIDITY.....	48
7.2.1	<i>Internal.....</i>	<i>48</i>
7.2.2	<i>External .....</i>	<i>49</i>
<b>8</b>	<b>CONCLUSION .....</b>	<b>50</b>
8.1	REJECTED HYPOTHESES .....	50
8.2	RECOMMENDATIONS. ....	50
<b>9</b>	<b>FUTURE WORK.....</b>	<b>52</b>

<b>10</b>	<b>ACKNOWLEDGEMENTS .....</b>	<b>53</b>
<b>11</b>	<b>BIBLIOGRAPHY.....</b>	<b>54</b>
	<b>APPENDIX A: EXAMPLE MAPPINGS USED IN THIS RESEARCH .....</b>	<b>57</b>
	<b>APPENDIX B: RELATION QUERYING BEHAVIOUR.....</b>	<b>58</b>
	ONE TO ONE (NON-RECURSIVE) .....	58
	ONE TO ONE (RECURSIVE) .....	63
	ONE TO MANY (NON-RECURSIVE) .....	64
	ONE TO MANY (RECURSIVE) .....	68
	MANY TO MANY (NON-RECURSIVE) .....	71
	MANY TO MANY (RECURSIVE) .....	71

## 1 Abstract

When an object oriented written application has to make use of a relational database an ORM tool can be used to synchronize the object model with the tables in the database. An advantage of such an ORM tool is the support for automatic querying of related objects. This option allows the programmer to query one object and dereference their relationships to other objects without querying each reference manually. The ORM tool will perform this relation querying automatically for the programmer.

A popular ORM tool for Java is Hibernate. In this research we focus only on Hibernate and in particular on the querying of related objects. Configuring the Hibernate configuration files can control this querying of related objects.

As the mapping of Hibernate (or any ORM tool) can be configured in many ways, poor performance can be a result of “wrong” configuration. Therefore we investigated the effect these configurations will have on the performance.

## 2 Introduction

Hibernate is one of the major ORM (Object Relation Mapping) tools written in and for Java. An ORM tool handles the persistence of objects in memory to a relational database. One of the main tasks of this is retrieving objects from the database. A key feature of this task is the automatic retrieving of referenced objects (of the queried object) as well, making it possible to dereference the references through an entire object graph without specifying the retrieval of each related object programmatically.

Hibernate supports a lot of different mapping configurations for customizing this behaviour, blurring the “best” performing choice in a specific situation. The mapping configurations describe the object to table mapping and relation querying behaviour. Especially, relatively new users (to the Hibernate library) can make unknowing choices that will drastically deteriorate the performance of the tool, maybe causing the abandoning of its usage in total.

Therefore, the goal of this research is to help the programmer find the best performing mapping configuration. To realize this goal we investigate the query behaviour of Hibernate for each type of mapping configuration. Interpreting this behaviour into best performing choices can still be a difficult task. We therefore benchmark certain configurations as well.

This research will be led by the following research question and sub-research questions.

### Research question

*What is the effect on the performance of Hibernate querying objects when using different mapping configurations?*

### Sub-research question 1

*What are the mapping configurations possibilities?*

### Sub-research question 2

*What is their effect on the behaviour?*

### Sub-research question 3

*What is the effect of the behaviour on the performance?*

The structure of the rest of this thesis is as follows. First we will discuss related work in chapter 3. After that we will give more detailed information about Hibernate and the mapping configurations in chapter 4, answering sub-research question 1. Then we analyse the querying behaviour of the mapping configurations in chapter 5, answering sub-research question 2. In the research method in chapter 6 we describe how we will perform our performance measurements and will describe and discuss the results in chapter 7, answering sub-research question 3. Finally we draw our conclusions in chapter 8, and indicate future work in chapter 9.

### 3 Related work

In [1] van Zyl et al. performed research on the performance of Hibernate compared to several other persistence techniques (like plain JDBC and object databases). They demonstrated the bad performance of Hibernate compared to these other techniques. In later research [2] they reviewed their implementation of Hibernate with a member of the Hibernate team, after several comments on the implementation by the Hibernate community. In this research they concluded that the performance gains when implementing the recommendations made by this member. They did not, afterwards, draw any conclusions on the performance of the renewed Hibernate implementation compared to the other persistence techniques. These recommendations are based on experience and rules of thumb.

We tried to improve their research and other research performing benchmarks with Hibernate by researching the effects on the performance of certain configurations available in Hibernate. With this knowledge we also try to help a programmer or researcher to implement a good performing Hibernate configuration for a specific situation.

Further research on the remaining configurations and possibilities of Hibernate and expanding it to other tools can help create more honest benchmarks in total.

#### 3.1 Patterns and techniques

There are a great variety of patterns and techniques to achieve a good performing ORM and Hibernate implements several of these. W. Keller described a complete pattern language on this subject in [3, 4, 5]. Several others also described best practices, patterns and techniques (often also implemented by Hibernate) in [6, 7, 8, 9]. Applying these patterns and techniques can be achieved by adjusting the mapping configuration of Hibernate.

As demonstrated in [2] and the remainder of this thesis, this configuration of Hibernate can have great influence on the performance. Due to the great variety of possibilities and the needed in depth database knowledge, this configuration remains a difficult and error-prone task that can easily deteriorate the overall performance. Due to this fact several have tried to automate this task, by analyzing the runtime behaviour [10, 11], static code analysis [12, 13] or by model driven generation [14], but none supplies a fully functional solution yet.

We decided to clarify the behaviour of Hibernate and its effect on the performance using these possible configurations for tackling this problem manually.

#### 3.2 Performance measurements

Measuring the performance of these persistence tools can be done by specialized benchmarks like: OO7 (java variant) [1], PolePosition and Torpedo [15]. The OO7 benchmark was originally intended to test OODBMS (Object Oriented Database Management Systems) [16]. In [1] this benchmark is translated to Java and used to compare the performance of Hibernate (and PostgreSQL) with db4o<sup>1</sup>, by van Zyl et al. indicated in the first part of this chapter. The same author(s) also wrote PolePosition, to further investigate the findings of this (translated) OO7 benchmark. In our research we used PolePosition as the basis for our benchmark.

---

<sup>1</sup> db4o is an object database created by Versant corporation.

Also it should be noted that [17, 18] wrote an ORM benchmark (“BenchORM”) with tests based on two real-life scenarios (the JDBC wrapper and the objects of the object model of this research were used in our tests).

### **3.3 The object-relational impedance mismatch**

Benchmarking persistence tools fit in a greater research, finding the optimal solution of solving/bridging the gap between the object oriented and relational paradigm. An ORM tool (like Hibernate) tries to fill this gap. This gap is also often referred to as the object-relational impedance mismatch [19].

As Robert Green stated in [20], the problem of “impedance mismatch” materializes itself strongly in two fundamental ways. One way is regarding the development burden presented by the mismatch and the other is regarding the slower performance and/or resource consumption imposed. As standardization of these mapping has occurred over the last couple of years, the development burden has been decreased and productivity has improved. Also the issues of performance and resource utilization have been relieved. By solving the “impedance mismatch” with ORM tools the developer is equipped to create a good performing solution easier.

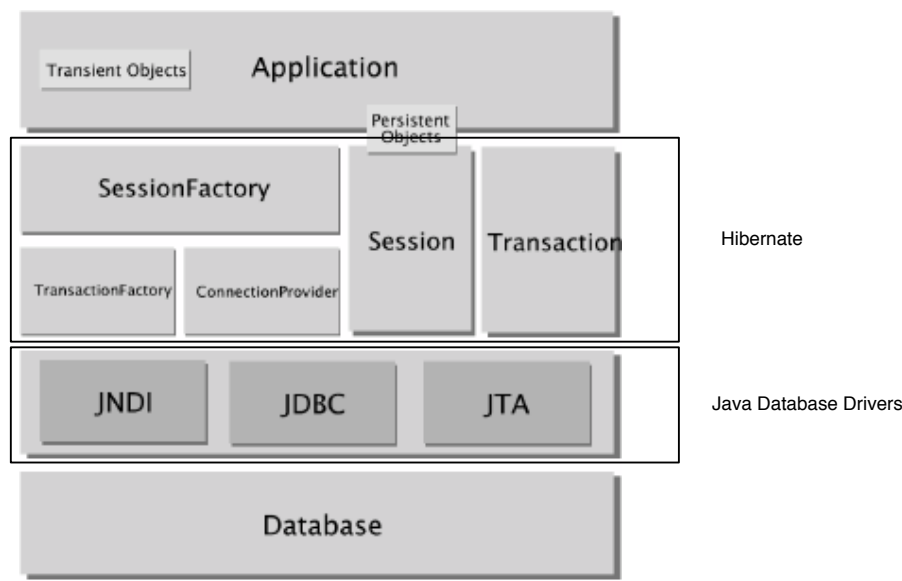
The use of object databases can also be an alternative since they have proven to be a better solution for certain kinds of applications [20]. However, this is out of scope of this research.

## 4 Introduction to Hibernate

In this part we will give some general information about Hibernate and will discuss the automatic retrieving of referenced objects (relation querying) in more detail.

Hibernate is an ORM (Object-Relational Mapping) library written in and for JAVA. An ORM is a communication layer between the application and the database (making use of the database drivers to send SQL statements), mapping an Object Oriented data model to a relational data model and handling the synchronisation of the two. For an overview of Hibernates architecture see Figure 1.

As both relational and object oriented paradigms are used, the terminology will slightly overlap. We speak of references when there is a relationship between objects and of relations when there is a relationship between tables.



**Figure 1: Context diagram of components and layers in the Hibernate Architecture (source [21]).**

In this research we focus only on the part where objects are queried from the database and will describe the other actions (like updating, deleting and inserting) in future work. Querying an object is the basic action also needed to perform the other actions of an ORM.

The Querying of objects can be initiated in two ways, by using the methods of the Hibernate API (Application Programming Interface) or dereferencing a reference. When dereferencing a reference the programmer has used the pointer in an object (retrieved by a method of the API) to access properties of the related object. When the API is used, the programmer has thus called a method that retrieves an object (or collection of objects). Objects retrieved with the API can be seen as the starting objects from which the dereferencing can begin.

The use of the API and the way these references are configured can have a great impact on the performance. From this API we only use one method (the `.get()`), the remaining methods are discussed in chapter 9 Future work. After using the method Hibernate will retrieve all referenced objects automatically. The “when” and “how” of this action can be configured by setting the (so called) fetching strategies and table representation. The object containing the reference will be referred to as the owning object from now on.



## 4.1 Relation querying configuration principles

Hibernate supports several configurations to determine the behaviour of querying related objects from the database: the fetching strategy and the table representation.

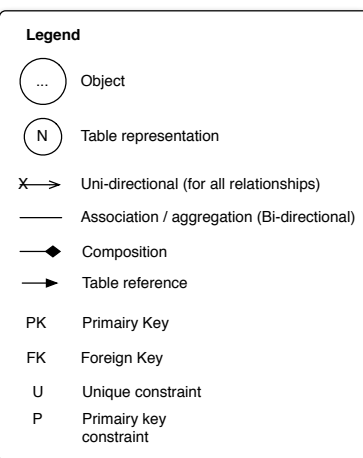
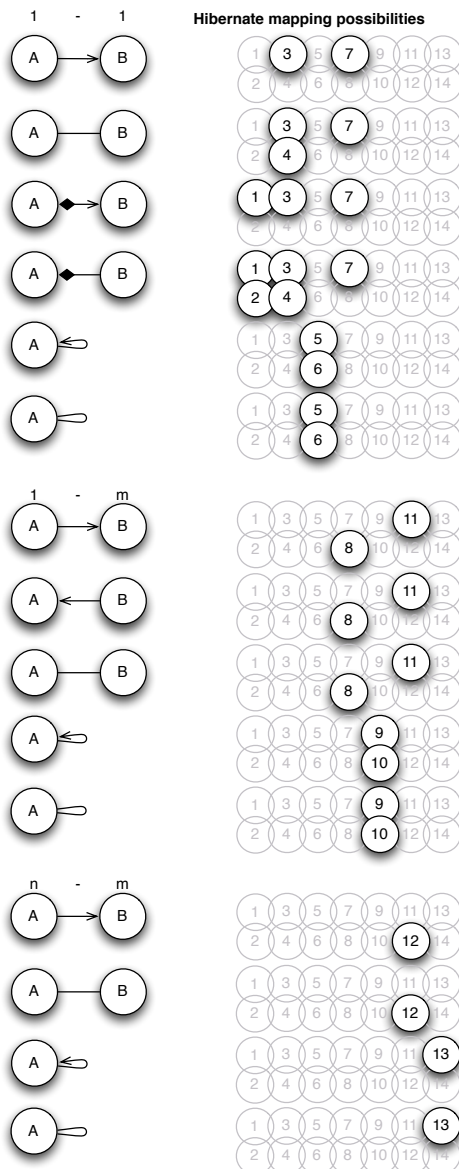
In the Hibernate reference documentation [22] a distinction is made between four fetching strategies and setting the laziness of it. When a reference to an object is set to lazy, Hibernate will wait with the retrieval of this object until it is first accessed. For each strategy we quote the definition given by the Hibernate reference documentation [22].

- **Select Fetching:** A second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you access the association.
- **Join Fetching:** Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
- **Subselect Fetching:** A second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you access the association.
- **Batch Fetching:** Hibernate retrieves a batch of entity instances or collections in a single SELECT by specifying a list of primary or foreign keys.

The SubSelect and Batch Fetching strategy are performance optimizations of the Select Fetching strategy. They can only be applied to one-to-many and many-to-many relationships and will have effect only when the owning object is in a collection. We scoped our research to only the Join and Select Fetching strategies.

The second influence on the behaviour is the table representation. There will be more queries or joins when the amount of tables the objects are divided over increases. Hibernate supports 13 table representations; we created an overview of the possibilities in Figure 2. This diagram contains at the leftmost column the object relationships (that we investigated) and a number next to it that indicates how this relationship can be mapped to tables indicated at the rightmost column. Also only primary and foreign keys are shown in this diagram, the other (value) columns are left out.

## Object reference representation



## Table representation

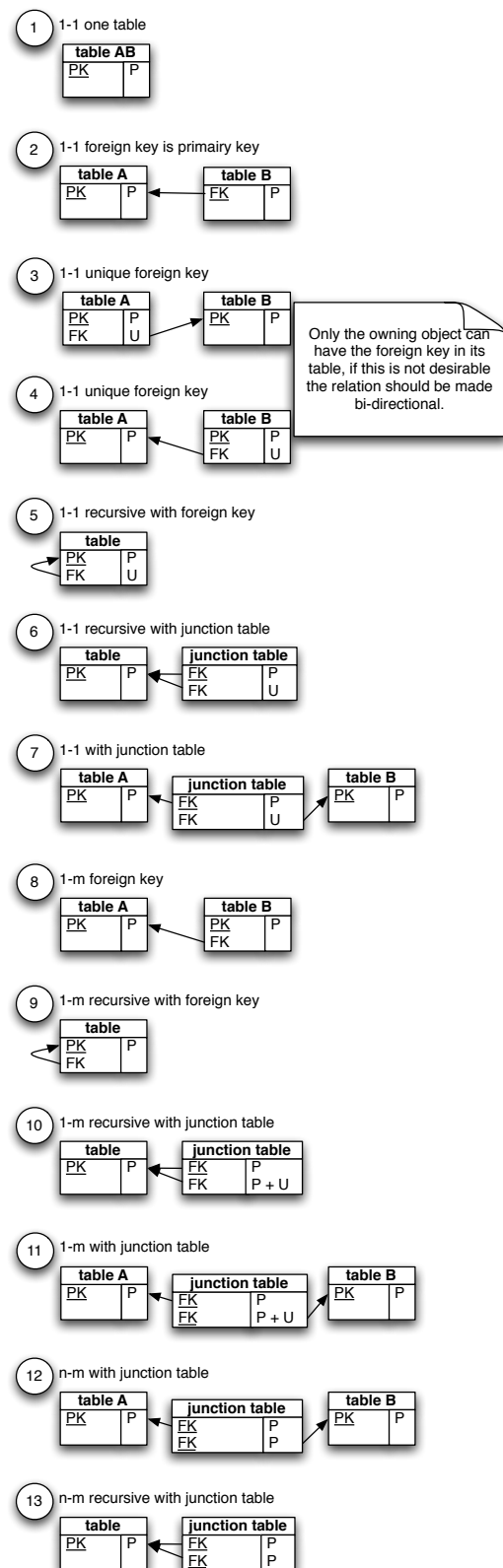


Figure 2: Mapping objects and relationships

## 4.2 Mapping configuration

In the next part we will describe how references to other objects can be configured in Hibernate.

The mapping configuration of Hibernate is done in XML files, for an example see Code fragment 1. A good practise is to have a XML file for each Java class. With the introduction of annotations in Java 5, Hibernate now also supports configuring the mappings with annotations. In our research we performed the mapping configuration in XML files.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.polepos.teams.hibernate.data">

    <class name="Person" table="person">

        <id name="id" column="id" type="long">
            <generator class="native"/>
        </id>

        <property name="firstName" column="firstname" length="40" />
        <property name="lastName" column="lastName" length="40" />
        <property name="age" column="age"/>

        <many-to-one name="adress"
            column="adressId"
            unique="true"
            not-null="false"
            lazy="proxy" fetch="select"
            cascade="all"/>

    </class>

</hibernate-mapping>
```

**Code fragment 1: Hibernate XML mapping file**

When configuring the mappings we distinguish between two definitions, one describing the type of relationship and one describing how this type is eventually configured in the mapping documents (type of mapping configuration). Namely, there are four types of relationships that can be configured by six types of mapping configurations.

The four types of relationships are: one to one, one to many, many to one and many to many. There are six Hibernate mapping configuration types and a range of attributes to configure them. The types of mapping configuration indicate either a reference to a single object or to a collection of objects. Using these types in different combinations, for both objects, can create the four types of relationships.

*Mapping configuration type for configuring a reference to a single object:*

- component
- many-to-one
- one-to-one
- join

*Mapping configuration type for configuring a reference to a collection of objects:*

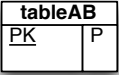
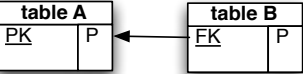
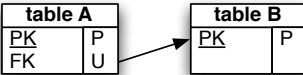

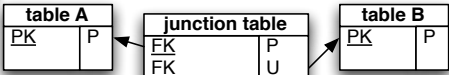

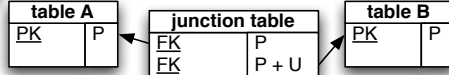
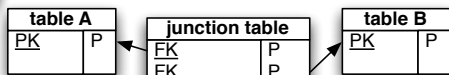
- one-to-many
- many-to-many

In the next part we first describe how to apply the type of mapping configurations and after that how to apply the fetching strategies.

### **4.3 Applying the type of mapping configuration**

In Table 1 we demonstrate how to create the table representations supported by Hibernate by listing the type of mapping configurations and key attributes (we used the XML notations for clarity). Note that the recursive table representations are equal to their non-recursive variant, but with a reference to their own class (we leave them out of the table to create a compacter overview).

Each Java object knows only about its own relationships. Configuring one or both objects of the relationship will therefore have different effects. Besides making the relationship bidirectional or unidirectional, it is also possible that some table representations cannot be created when configuring only one object. The optional sign will indicate that the relation can be unidirectional.

Table Representation (TR)	Configuration object A	Configuration object B
<p>1 1-1 one table</p> 	<component> <sup>2</sup> with the properties of B within the start and closing tag.	No xml-mapping document needed. For a bidirectional relationship, add the <parent> element tag within the <component> tag of object A.
<p>2 1-1 foreign key is primary key</p> 	<one-to-one>	<one-to-one constrained="true">
<p>3 1-1 unique foreign key</p> 	<many-to-one unique="true">	<b>(optional)</b>  <one-to-one>
<p>4 1-1 unique foreign key</p> 	<one-to-one>	<many-to-one unique="true">
<p>7 1-1 with junction table</p> 	<join> <key> <many-to-one unique="true">	<b>(optional)</b>  <join> <key unique="true"> <many-to-one>
<p>8 1-m foreign key</p> 	<set> <one-to-many>	<b>(optional)</b>  <many-to-one>
<p>11 1-m with junction table</p> 	<set> <many-to-many unique="true">	<b>(optional)</b>  <join> <key> <many-to-one>
<p>12 n-m with junction table</p> 	<set> <many-to-many>	<b>(optional)</b>  <set> <many-to-many>

**Table 1: Type of mapping configurations**

<sup>2</sup> The <component> element maps properties of a child object to columns of the table of a parent class.

In “Appendix A: Example mappings used in this research” we listed the relationship configuration we used in our test setups.

For more detail explanation see the Hibernate reference documentation [22].

#### 4.4 Applying the fetching strategy

Each type of mapping configuration can be set to a specific fetching strategy, by setting three attributes (the possible values of the attributes are described below this list):

- the lazy attribute: specifying when the reference needs to be retrieved (default: true|proxy);
- the fetch attribute: specifies the query used to retrieve the reference (default: select);
- the batch-size attribute: specifying of how many objects (in a collection) the relationship needs to be retrieved (default: 0).

The fetching strategy attributes will have no effect on the join and component mapping configuration types, because the join type will always force a join and the component type will force the storage of two objects into one table (making it not possible to join or perform separate select queries).

Not each one of the remaining configuration types can contain all these attributes. In the next list we created an overview of the possibilities (taken from the hibernate reference documentation [22]).

- one-to-one
  - o fetch="join|select"
  - o lazy="proxy|no-proxy|false"
- many-to-one:
  - o fetch="join|select"
  - o lazy="proxy|no-proxy|false"
- set:
  - o fetch="join|select|subselect"
  - o lazy="true|extra|false"
  - o batch-size="N"
- many-to-many:
  - o fetch="select|join"
  - o lazy="true|extra|false"

For more detail see the Hibernate reference documentation [22].

## 5 Theory of Hibernate's query behaviour

As Hibernate communicates by sending SQL statements, the logical first step is to examine and create an overview of the executed SQL statements. With this overview we will form our query behaviour theory. Analyzing this theory will indicate the different factors that change when different mappings are applied, but not their effect on the performance. Therefore we will form several hypotheses based on this theory and execute performance measurements to test them in chapter 7.

We created this theory by examining the SQL queries executed by Hibernate when performing the following tests. We created for each mapping possibility a test containing objects (that implement the mapping configuration and fetching strategies). Then we retrieve the owning object and dereference the reference(s), while registering the executed SQL using Hibernate's logging feature.

The theory explained in this chapter is the need to know basis for creating the query behaviour for every type of object model.

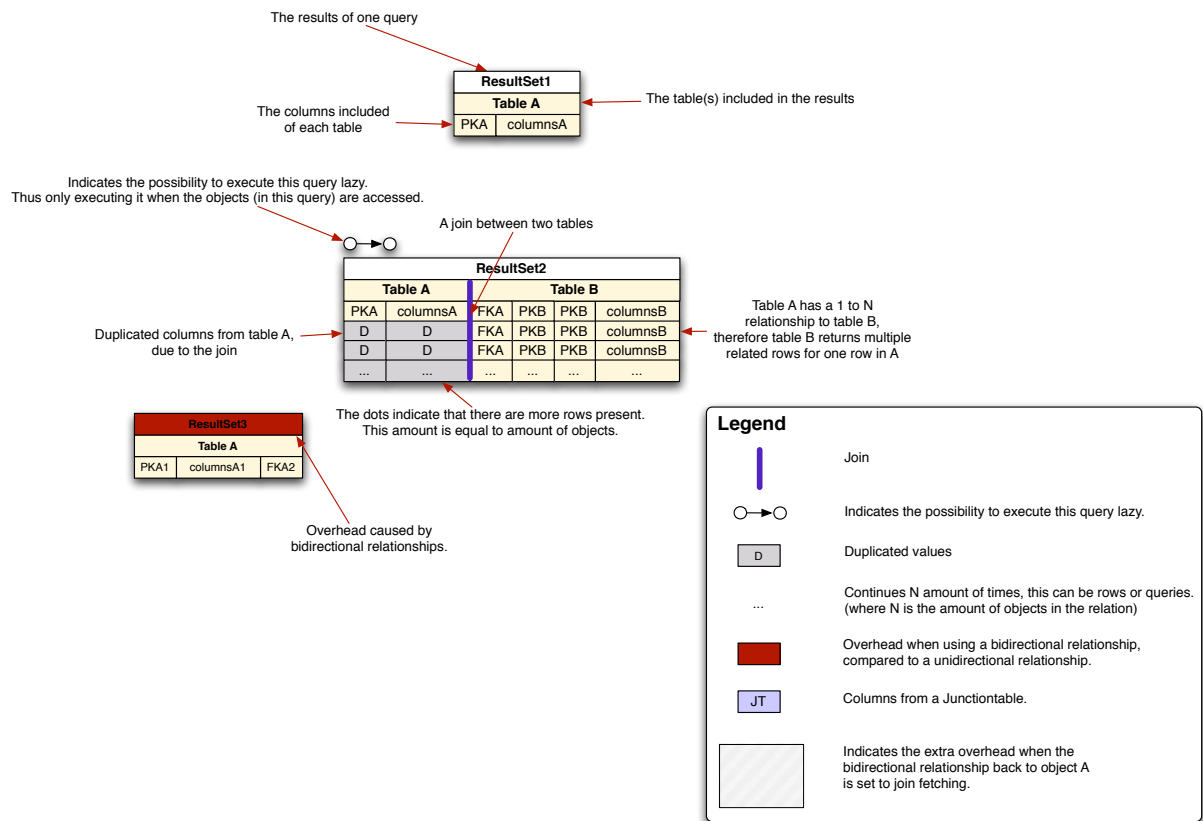
First we describe our notation and the basic behaviour of each mapping and after that we will describe the influences of the following configurations on the basic behaviour:

- a bidirectional relationship;
- lazy configuration;
- join fetching;
- recursive relationships; and
- concatenating relationships

For a complete understanding we worked out the behaviour for all relationships (one-to-one, one-to-many and many-to-many) in Appendix B: Relation querying behaviour. These examples also demonstrate the behaviour of recursive and bidirectional relationships.

### 5.1 A notation for query behaviour

We present the SQL in the form described below. As SQL visualized in form of text will not contribute to the clarity of the overview, we decided to use a more visual representation of the result sets instead. In that representation we show the actual results in form of a table, leaving out the more unimportant detailed information. Figure 3 explains our notation.



**Figure 3: Notation hibernate querying behaviour**

## 5.2 Basic behaviour

In the next examples there will be an object A with a reference to object(s) B. For both objects we describe what type of mapping configuration is applied to it. The ResultSets indicate what queries were executed when object A and related object B are retrieved. When predicting the behaviour of the reference from object B back to A, it is possible to concatenate these isolated building blocks together. In this case it should be noted that the rules described in the other subchapters of this chapter must be applied as well.

To make the mapping configuration type more visible, we enclosed them in <...>.

With the <component> configuration type all properties of object B are included in the select query of A. See Figure 4. In this table representation both objects are stored in one table.



**Figure 4: Query <component>**

With the <one-to-one> configuration type another query is executed to retrieve object B (by foreign key). See Figure 5 and Figure 6. Both objects are stored in separate tables and the foreign key is stored in the table of object B. When applying only the <one-to-one> configuration type the foreign



key is set as primary key in object B by default. Adding the <many-to-one> configuration type to the mapping of object B will make the foreign key a separate column in the table.



Figure 5: Queries <one-to-one>

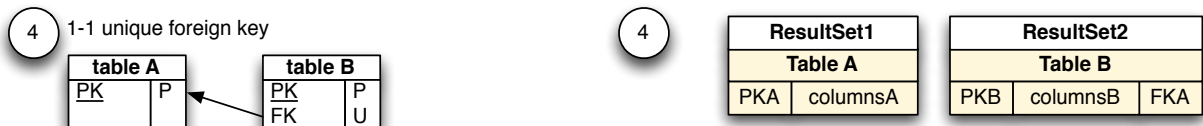


Figure 6: Queries <one-to-one> on A and <many-to-one> on B

With the <many-to-one> configuration type also another query is executed to retrieve object B (by primary key). See Figure 7. Both objects are stored in separate tables, but the foreign key is stored in the table of object A. This introduces the possibility to execute the query for object B lazy. Meaning that Hibernate can create a proxy object for B, because after the first query the ID of object B is known. This proxy object will retrieve object B as soon as one of its properties is accessed.

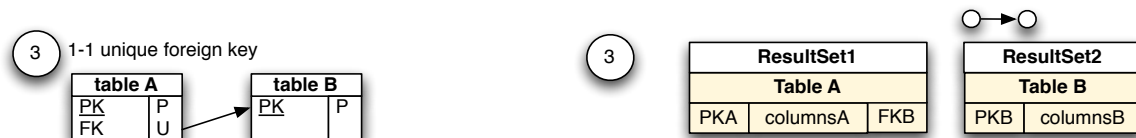


Figure 7: Queries <many-to-one>

With the <join> and <many-to-one> configuration type the foreign key is joined in the first query and for object B another query is executed (by primary key). See Figure 8. Both objects are stored in separate tables and the foreign keys are stored in a separate table as well. Each object will therefore always be retrieved by their primary key, instead of sometimes also by the foreign key. This situation makes it possible to always execute the second query lazy, while the previous configurations did not allow this.

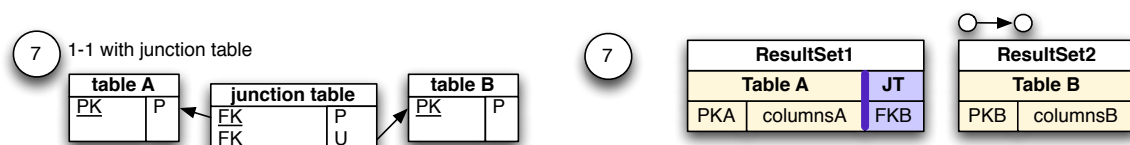


Figure 8: Queries <join> and <many-to-one>

With the <set> and <one-to-many> configuration type another query will retrieve all B's connected to A (by foreign key). See Figure 9. For this relationship Hibernate requires two mapping configuration types. The <set> indicates that the results belong to a collection were the <one-to-many> will indicate the type of relationship.

In this case it is possible to execute the second query lazy as well because the collection will function as proxy object (waiting with the execution of the query until the collection is accessed).

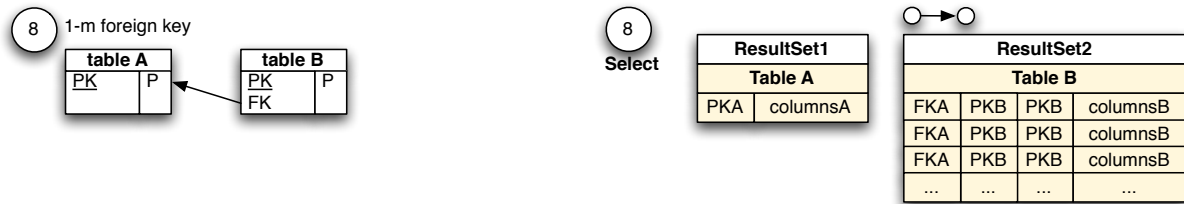


Figure 9: Queries <set> and <one-to-many>

With the <set> and <many-to-many> another query will be executed to retrieve the foreign keys from the junction table and then for each related object a query is executed to retrieve it (by primary key). See Figure 10. Not changing this basic behaviour can increase the amount of executed queries drastically when the related objects increases. **Note:** When a unique constraint is added to the <many-to-many> tag, the relation will be a one to many relationship.

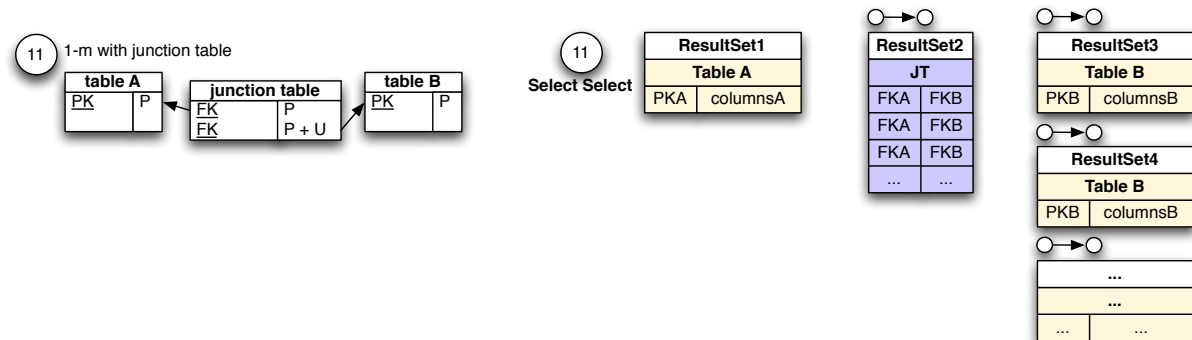


Figure 10: Queries <set> and <many-to-many> with unique constraint

With the <set> and <many-to-many> (without a unique constraint) the behaviour is similar to the previous one to many relationship. See Figure 11. **Note:** When no unique constraint is added, the relation will be a many to many relationship.

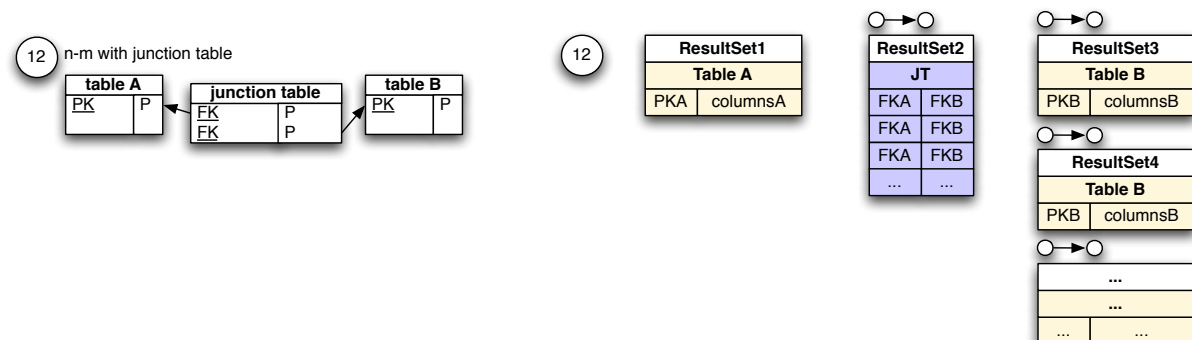


Figure 11: Queries <set> and <many-to-many>

## 5.3 Important factors

To understand the query behaviour of a certain object model there are some important factors that change the basic behaviour described in chapter 5.2. In the next part we describe these factors.

### 5.3.1 Bidirectional relationship

In a bidirectional relationship both objects have configured a reference to each other by using the normal configuration types. When processing these configuration types, Hibernate will act as the default behaviour described (in the previous chapter). What should be mentioned is that the default behaviour is always preceded by a session cache lookup that can reduce the amount of queries when the object is retrieved before. Theoretically this should mean that when the reference back is processed no extra queries will be executed, but due to the working of the session cache this is not always the case. When performing a cache lookup, the ID (primary key) of the object has to be known. In chapter 5.2 we indicated that an object was queried by primary or by foreign key, in the case the object is queried by the primary key Hibernate can retrieve the object from the session cache. If this is not the case Hibernate will perform an extra query (retrieving the first object again).

In the following part we will demonstrate this behaviour by giving an example. Here we will retrieve object A, dereference the reference to object B and dereference it back to object A. With the table representation of Figure 12 an extra query for the reference from object B back to object A will be executed (because the table of object B does not contain the ID of object A). In Figure 13 object A can be retrieved from the session cache.

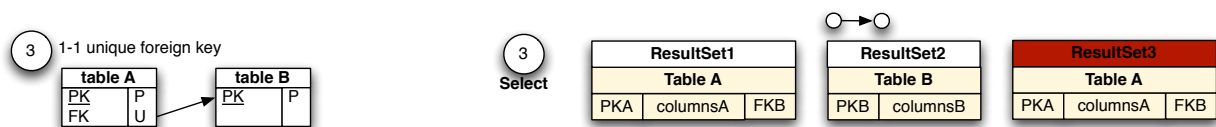



Figure 12: Object B cannot find object A in the cache



Figure 13: Object B can find object A in the cache

### 5.3.2 Lazy configuration

In 5.2 Basic behaviour we indicate with a  if the query of a referenced object can be executed lazy. This sign represents two objects and dereferencing the reference of the left object to the right object. In lazily configured references this dereferencing is the trigger for the query of the right object to be executed.

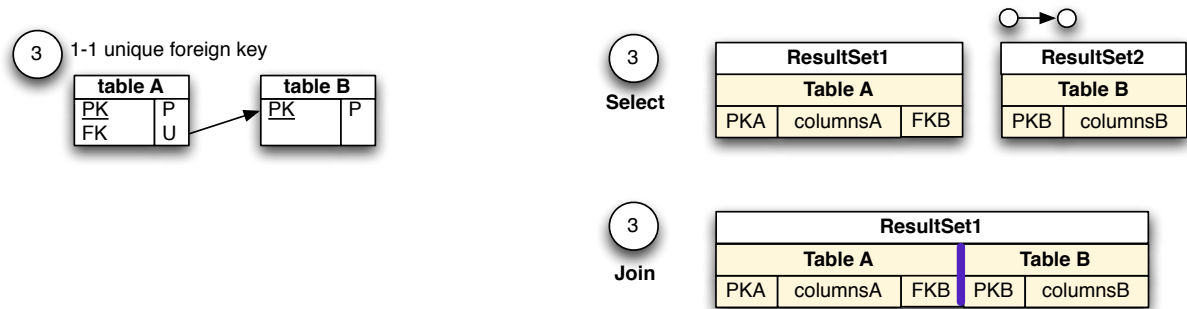
When configuring the laziness of a reference, Hibernate distinguishes between a reference to a single object or to a collection of objects. When processing a lazy reference to a single object, a proxy object is created (to handle the lazy behaviour). When processing a lazy reference to a collection of objects, no proxy object is needed (due to the collection object that will handle the lazy behaviour).

In case of a reference to a collection of objects the query can always be executed lazy but when relating to single objects this is not always the case. To create a proxy object, the ID (primary key) of the proxied object is needed. In 5.2 Basic behaviour we indicated for each table representation

whether they were queried by primary or by foreign key. For the relationships that are retrieved by foreign key, the ID will not be present at creation of the proxy object. Hibernate will therefore not be able to create a proxy object and the query is executed immediately.

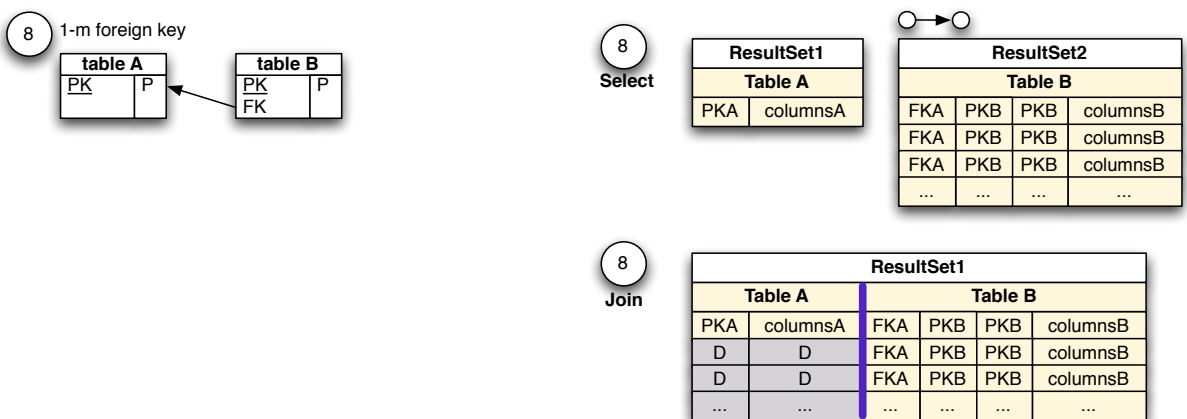
### 5.3.3 Join fetching

When a reference is configured to join fetching, Hibernate will join the query of the reference (to B) with the query of the owning object (A). See Figure 14.



**Figure 14: Effect of join fetching a relationship**

When the reference is an one to many relationship, the “one” side will be duplicated to fill up the amount of rows on the “many” side. See Figure 15.



**Figure 15: Effect of join fetching a one to many relationship**

There are several rules to keep in mind before this simple technique, to create the behaviour for joined queries, can be applied.

When there is a chain of referenced objects with each reference configured to join fetching, Hibernate will combine all these queries into one query by joining the tables of all the objects. This process (of joining the tables of each related object) will continue down the reference line until one of the following two causes is encountered: (1) when the maximum fetch depth is reached (configured in the general Hibernate configuration) or (2) when the object type is already joined in the query once. The latter reason can be the case when one type of object is related by two other

types of objects (for example: an employee and a customer both reference instances of the address object).

#### **5.3.4 Recursive relationship**

With a recursive relationship it should be noted that the referenced object will contain the same reference relationship with another object (of the same type) as the owning object. This can introduce extra queries.

This is best understood with a parent/child relationship. When the child has a reference to the parent and the parent will have a reference back to the child, it should not be forgotten that they are the same type of object. Therefore the parent is also a child and will have a reference to his parent as well and the child is also a parent and will have a reference to his child as well, this can cause Hibernate to execute extra queries.

#### **5.3.5 Concatenating relationships**

The behaviour of a specific object model can be created by concatenating the building blocks of “5.2 Basic behaviour” and obeying the “rules” of the previous subchapters.

## 6 Research method

In chapter 5 we described the theory of the relation querying behaviour of Hibernate. To choose the best performing mapping configuration using this theory requires still a decent amount of database knowledge. Therefore we constructed several performance tests that will indicate the performance differences between key behaviour differences in this theory.

Besides these behaviour differences there are also several other factors that can influence the performance for which we need to choose a value/implementation. In this chapter we will therefore first describe how we perform our tests and measurements, secondly what factors can cause differences in the performance and what values/implementations we choose for these and finally what we do to stabilize the environment.

### 6.1 Measuring the performance

When measuring the performance we had to set up an environment, a couple of tests and the measuring instrumentation. Each one of these is described in the next part of this chapter.

#### 6.1.1 Environment set up

To prevent that the benchmark and databases influence each other, we divided them over different desktops. We constructed the following environment to perform our tests (see Figure 16).

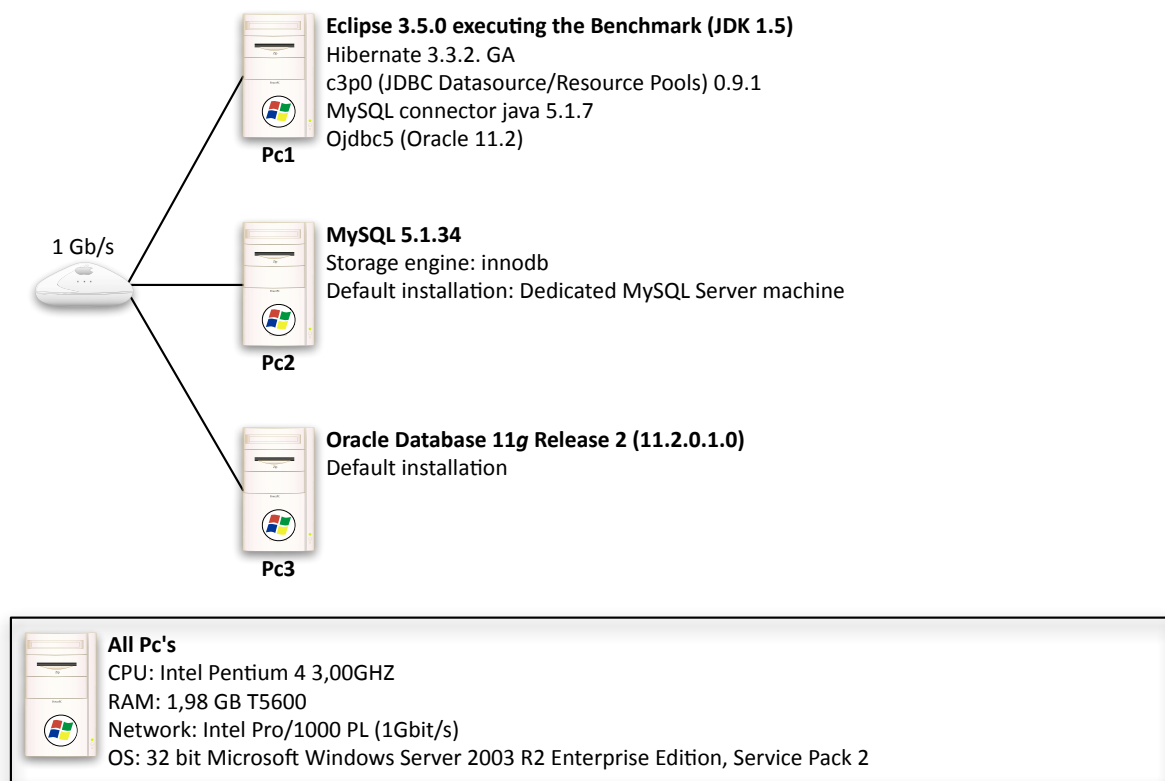
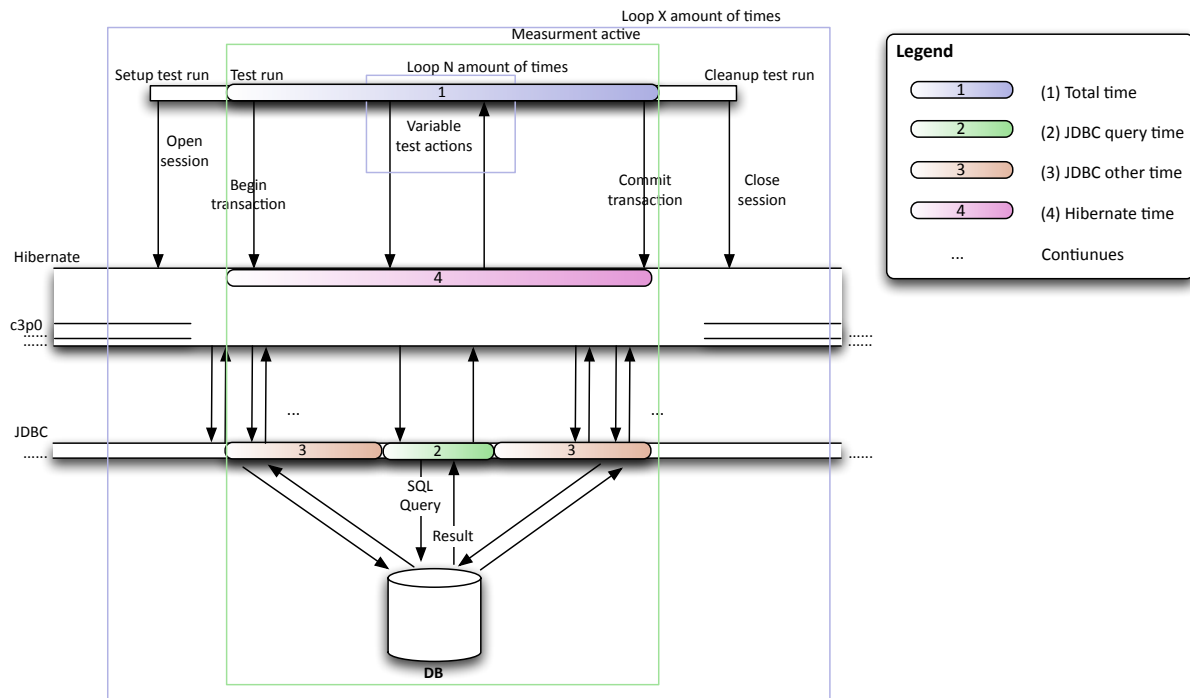


Figure 16: Environment

We used three similar desktop computers that each performs a specific task: one executing the tests (and Hibernate, c3p0 and JDBC libraries) and two each containing a database implementation.

### 6.1.2 Test set up

For our performance measurements we use the following test and measurement set up, see Figure 17. In this set up the “test run” will control the execution, calling methods of the Hibernate API. Hibernate will then perform the object relation mapping and will use the JDBC driver to execute the queries to the database.



**Figure 17: Test and Measurement environment**

The “variable test actions” represents retrieving the objects that have implemented the specific mapping configuration we want to test. Executing the “variable test actions” only one time will be immeasurable (with our equipment); we therefore repeat this action 30.000 times with a one to one relationship, 3.000 times with a one to ten relationship and 300 times with a one to hundred relationship. We reduce the amount of times due to the memory limitations (all retrieved objects stay in the session cache and because we don’t want to flush this cache this amount had to be reduced).

Executing this action in a smaller amount of times will also be measurable, but we choose for this amount because the greater the amount of repetitions the closer the average will be to the average of repeating it infinite (the entire population). Taking therefore into account as much fluctuations in the results as possible, giving a more representative view of reality.

Ideally, when we execute this test multiple times the measurement results of each repetition should be equal to each other. In our environment (and in most environments) this is not the case. We therefore investigated our results and noticed that the first run always contains a high deviation compared to the other runs and is therefore excluded (cold run). The other runs lie more closely together. Repeating this test four times will give a clear indication of the performance (of a particular setting) and we therefore choose 4 for the X in Figure 17. In our results we also indicate the standard deviation of these four runs.

### 6.1.3 Measurement set up

To prevent other processes to influence the measurements we measure only the time our test is active, thus by measuring the “CPU time”. The “CPU time” of a thread is the sum of the “User time” (time spent running the threads code) and the “System time” (time spent running operating system code on behalf of the thread). When the java virtual machine does initiate the Garbage collector, or any other process (as well as any other process initiated by the operating system) this will not effect the “CPU time”.

As the environment (see Figure 17) exists of roughly three separate functioning parts (hibernate, the JDBC and the database), we distinguish the time spent in each part separately. We measure the “Total time”, “JDBC query time” and “JDBC other time” to calculate the times spent in the separate parts. The “Hibernate time” is calculated by subtracting the “JDBC other time” and “JDBC query time” from the “Total time”.

In the next list we explain these times:

- *Total time*: The time from the start of the test run until the end, including the time executing in underlying layers.
- *JDBC query time*: The time executing the query; thus transfer over the network, gathering the results in the database and processing them into a ResultSet.
- *JDBC other time*: All other time spend executing the JDBC code; like starting/committing the transaction, creating the prepared statement (plus setting the parameters) and retrieving the result from the ResultSet.
- *Hibernate time*: The time spent executing Hibernate code.

Time spent executing the code of the benchmark can be neglected as these are only hibernates calls, a for loop and the assignments of some variables.

To perform the measurements in the JDBC driver, we use the JDBC wrapper created by André Calero Valdez and Firat Alagöz [17, 18].

## 6.2 Preventing bad performance one to many relationship in Oracle

When configuring a one to many relationship, the Oracle dialect (used in Hibernate to communicate with an Oracle database) does not create an index on the foreign key. This will deteriorate the performance of these relationships drastically. In all other situations (and also in the MySQL database) this index is created. In a forum threat [23] the Hibernate team indicated this should also be the case for the oracle JDBC, but until present day this is not yet adjusted.

Therefore we created, as a work around found in [24], a database-object in the mapping configuration of the object implementing the one to many relationship. In this database-object we manually specified the creation of the index on the foreign key:



```

<hibernate-mapping [...]>

[...]

<database-object>
<create>CREATE INDEX indexName ON objectName(columnName)</create>
<drop>DROP INDEX indexName ON objectName</drop>
<dialect-scope name="org.hibernate.dialect.OracleDialect"></dialect-scope>
</database-object>

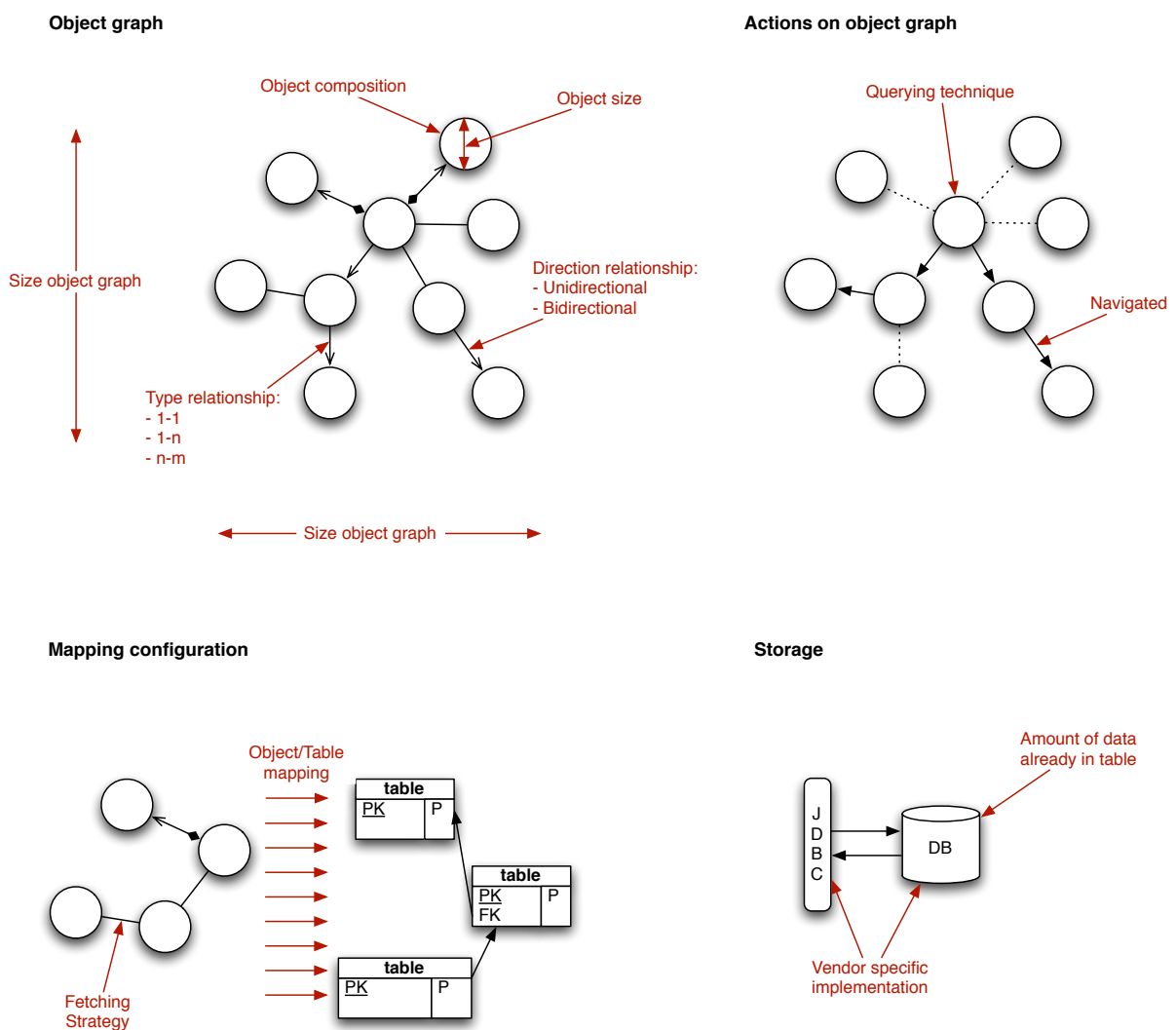
</hibernate-mapping>

```

**Code fragment 2: Creating an index for the foreign key in the Oracle database**

### 6.3 Factors influencing the performance

There are several factors influencing the performance when querying relationships. In this part we will discuss the factors and what standard values we choose for them. For an overview of the factors, see Figure 18. In this overview we left out the influence of several environment aspects as hardware (IO/network traffic), operating system and virtual machine.



**Figure 18: Performance influences of relation querying**

### 6.3.1 Object graph

The type of relationship influences the performance by forcing a specific table representation. This is closely linked with the table representation and all types of relationships need to be tested in order to test all table representations. It also does matter whether a relationship is unidirectional or bidirectional because in some situations extra queries can be executed.

Depended on the size of the object graph (a set of related objects within an object model), size of the objects and composition it will take more time to transfer and progress all the needed objects and properties.

Each test has its own object graph. We choose an object graph existing of two type of objects (and one for recursive relationships) that have a relationship with each other. For the object composition we use the object model from [17, 18]. In their research they investigated two real-life scenarios and created a benchmark depending on these scenarios. The scenarios also described an object model that could be translated to objects of the following size and composition (we call base objects). The base objects are flat (no relationships) objects containing only strings and integers (and a long as ID). For each test the base objects can be extended with a relationship to another one of these base objects.

The objects are composed of the following value types:

- Long: the identifier (called "ID"), every objects has an ID;
- String: a property
  - o Smallstring: with a maximum of 40 characters.
  - o Bigstring: with a maximum of 4000 characters.
- Integer: a property.

We distinguish 5 base objects with different compositions of the values described above (we will also refer to the numbers in front of these objects instead of their names):

- O1. FlatSmallObjectSmallString: Object with 1 property, a smallstring.
- O2. FlatSmallObjectInt: Object with 1 property, an integer.
- O3. FlatSmallObjectBigString: Object with 1 property, a bigstring.
- O4. FlatBigObjectSmallString: Object with 50 properties, all of type smallstring
- O5. FlatBigObjectInt: Object with 50 properties, all of type int.

### 6.3.2 Mapping configuration

By configuring the relationship mappings Hibernate will differentiate in query behaviour. For our performance tests we choose those mapping configurations that differentiate in these behaviour differences. The key differences in this behaviour that we are trying to measure are:

- the amount of queries: by dividing the objects over one table or joining two queries together;
- the costs of joining: for both objects table and junction tables;
- not retrieving an object;
- the amount of duplicated values: caused by joining;
- the overhead: caused by bidirectional relationships.

We translated these to hypotheses discussed in chapter 7 Performance measurements.

### 6.3.3 Actions on object graph

How the first object is retrieved from the database and whether dereferencing a reference influences the performance. Some querying techniques perform worse than others and when a reference is set to lazy and not dereferenced, the query retrieving the referenced object will never be executed.

When retrieving an object we use the Hibernate get method. The other object query techniques are described in chapter 9 Future work. Each reference is always dereferenced, unless it is described in the hypothesis that it is not.

### 6.3.4 Storage

How a vendor implements their database (and corresponding JDBC driver) can have great influence on the overall performance. As Hibernate depends on the database to function, we choose to use two different databases for our tests. Also the amount of data that has to be searched will influence the overall performance and is therefore kept equal in all compared tests.

We use the following databases: Oracle 11g and MySQL Community Server 5. For the amount of data present in the tables, we choose to only store the objects needed for the test (before starting the tests). This amount will never be greater than 30.000 for a table.

## 6.4 Stabilisation

To stabilize the environment we choose the configurations of the tests to be in tune with the limitations of the memory of the environment, preventing unnecessary performance disturbance when crossing this limitation (like early precautionary measures taken by the operating system). We also turned off the windows page files, preventing the early use of the hard disk when memory starts reaching its limits.

As anyone who performs a benchmark we disabled as many features/processes (of the operating system) as possible that do not have any relevance for the measurements we want to take, trying to prevent accidental high processor utilizations.

To prevent caching done in underlying layers and guarantee that no run can benefit from another run, the object values (the data transmitted over these layers) are randomly generated. This will make the probability of an object to be unique increasingly large. Also the caching of queries is turned off in the database.

To prevent housekeeping tasks of the JVM (Java Virtual Machine) to be performed during the tests as much as possible, we execute the garbage collector after each run. During this call the control of execution is temporary given to the garbage collector that will make a best effort to reclaim space from all discarded objects (it is possible that the garbage collector will not reclaim space, but with this set-up it is given time explicitly and time outside the measurements). Only when control is returned to the benchmark the next test will be run. Also the maximum to which the heap size of the JVM can be increased is set to 1300 MB preventing unnecessary garbage collection during the execution of the tests and the starting heap size is directly set to the maximum (preventing the heap size to be increased during the test runs). To establish this, the JVM is configured to start by using the “-Xms1300m -Xmx1300M” flags. We also run the benchmark with the server compiler, using the “-server” flag as program argument.

## 7 Performance measurements

In this chapter we describe the results of several executed performance measurements.

Analyzing the behaviour, described in chapter 5, we observed three changing factors: the amount of queries, the type of query and the size of the results. We created hypotheses based on these changing factors. Each hypothesis contains a diagram explaining the query behaviour (of the executed test), a diagram displaying the results and a summary table of the results.

In the next part we state the hypothesis and display the results of the performance measurement, but first we briefly summarize all hypotheses in the following list:

- 1) **Hypothesis:** Retrieving two objects from one table will perform faster than retrieving two objects from two tables, using a join query.
- 2) **Hypothesis:** Storing the foreign key within the objects table will perform faster than storing it in a junction table.
- 3) **Hypothesis:** Retrieving two objects from one table will perform faster than retrieving two objects from two tables, using two select queries.
- 4) **Hypothesis:** Retrieving two objects from two tables with a join query will perform faster than retrieving them using two select queries.
- 5) **Hypothesis:** Retrieving one object from one table will perform faster than retrieving two objects using a join query.
- 6) **Hypothesis:** Retrieving an one to many relationship (with 10 objects on the many side) will perform faster when the relationship is set to select fetching than to join fetching.
- 7) **Hypothesis:** Retrieving an one to many relationship (with 100 objects on the many side) will perform faster when the relationship is set to select fetching than to join fetching.
- 8) **Hypothesis:** Join fetching the junction table with the many side will perform faster than select fetching each object from the many side, with an amount of 10 objects at the many side.
- 9) **Hypothesis:** Join fetching the junction table with the many side will perform faster than select fetching each object from the many side, with an amount of 100 objects at the many side.
- 10) **Hypothesis:** Storing the foreign key within the objects table will perform faster than storing it in a junction table, with a one to many relationship and an amount of ten objects at the many side.
- 11) **Hypothesis:** Using a mapping configuration that does not introduce overhead when configuring a bidirectional relationship will perform faster than using a mapping configuration that does.

## 7.1 Hypotheses and results

In this subchapter we will state the hypotheses again and display the results of the performance measurement.

### 7.1.1 Testing the type of queries

- 1) **Hypothesis:** Retrieving two objects from one table will perform faster than retrieving two objects from two tables, using a join query.

Accepted

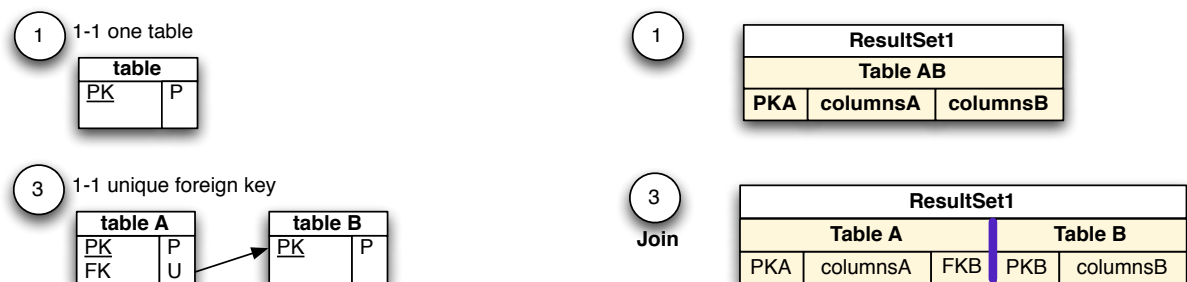


Figure 19: Query behaviour hypothesis 1

In this test we related two objects (of the same type) 30.000 times with each other, using the two mapping configuration described above. See Figure 20 and corresponding summary Table 2.

We will refer to the first mapping configuration (in this case: storing two objects in one table) as configuration one and to the second mapping configuration (in this case: storing two objects in two tables, relating them with an unique FK and retrieving them with a join query) as configuration two.

When observing the results of the tests with the Oracle database, in all situations configuration one performs better than configuration two. Comparing these results with those of the tests with the MySQL database, we see that with the large objects there is hardly any difference between the two methods.

It can also be noted that the time spent executing in the database and transferring the data (the JDBCQuery time) stays relatively low when the object size increases, while the JDBC time drops in performance drastically. Recall that O4 and O5 are 50 times larger than O1 and O2 (for an explanation of the different type of objects, O1 till O5, see 6.3.1 Object graph).

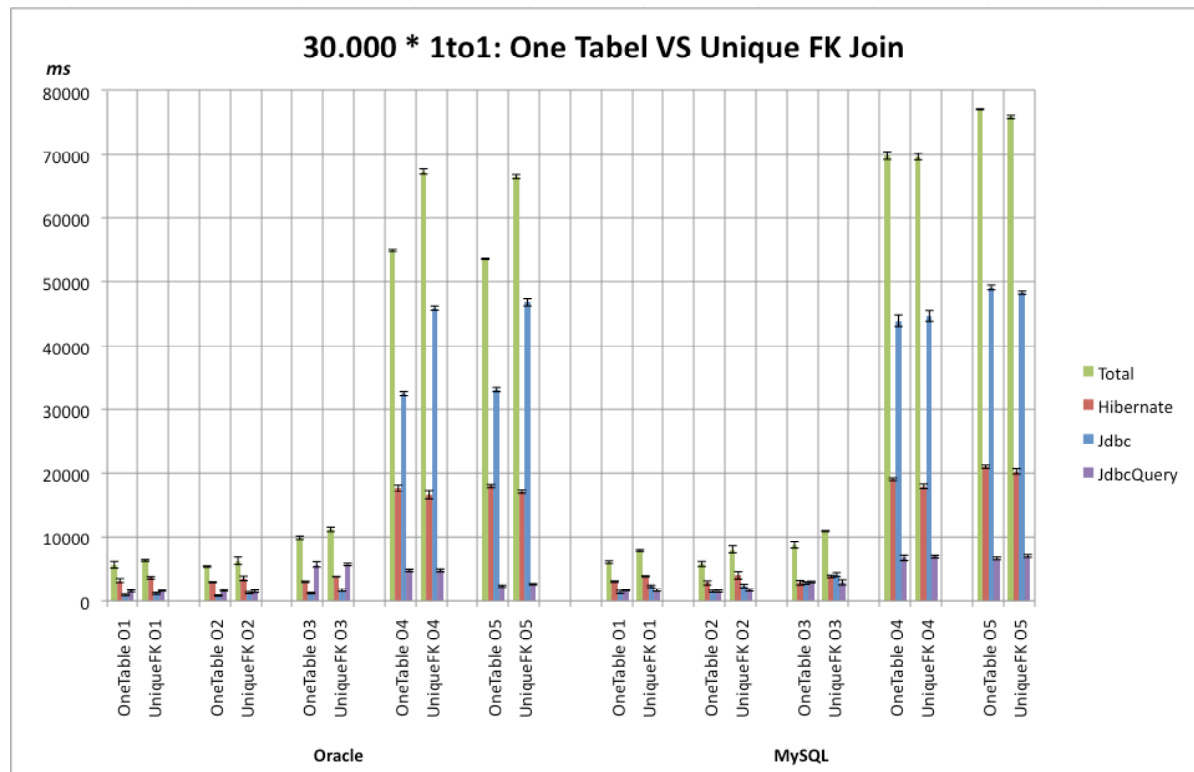


Figure 20: Performance measurements hypothesis 1

The results diagrams (like Figure 20) describe the three measured values and indicate the standard deviation at the top. The objects (of the object model) are indicated with O1 to O5. To keep the names (under the bars) as short as possible, we also choose to only describe the key differences in this name.

In Table 2 we indicated how much slower the second configuration is compared to the first configuration, by indicating the difference in percentages (where configuration one is the total). When the percentage is negative, configuration two performed faster and when positive configuration one did. We also display the highest standard deviation of the two compared configurations, in percentage (calculated using the first configuration as total).

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	11,8%	9,1%	13,2%	9,3%	24,8%	15,7%	1,3%	11,3%
O2	16,7%	10,8%	19,8%	11,4%	56,3%	17,7%	-8,6%	12,1%
O3	12,7%	3,3%	26,1%	3,4%	41,0%	16,6%	-0,2%	7,3%
O4	22,4%	0,7%	-5,8%	4,0%	41,1%	1,2%	0,2%	4,6%
O5	24,1%	0,6%	-5,2%	1,7%	40,8%	1,7%	13,0%	6,9%
MySQL	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	29,7%	3,4%	26,3%	3,5%	68,3%	16,6%	3,1%	12,0%
O2	38,9%	9,5%	41,5%	20,1%	59,0%	19,9%	14,2%	12,7%
O3	25,5%	5,7%	34,2%	11,9%	45,8%	10,7%	-1,9%	13,1%
O4	-0,2%	0,9%	-5,7%	1,8%	1,7%	2,0%	2,9%	6,1%
O5	-1,5%	0,3%	-3,5%	2,0%	-1,7%	0,8%	5,5%	3,2%

Table 2: Summary performance measurements hypothesis 1

## Conclusion

The performance between retrieving two objects from one table or retrieving them from two tables with a join query is either better or neglectable when using one table. The performance is neglectable with large objects and the MySQL database. Therefore using one table to store two objects will (mostly) perform better than using two tables and retrieving the objects with a join query.

- 2) **Hypothesis:** Storing the foreign key within the objects table will perform faster than storing it in a junction table.

Rejected

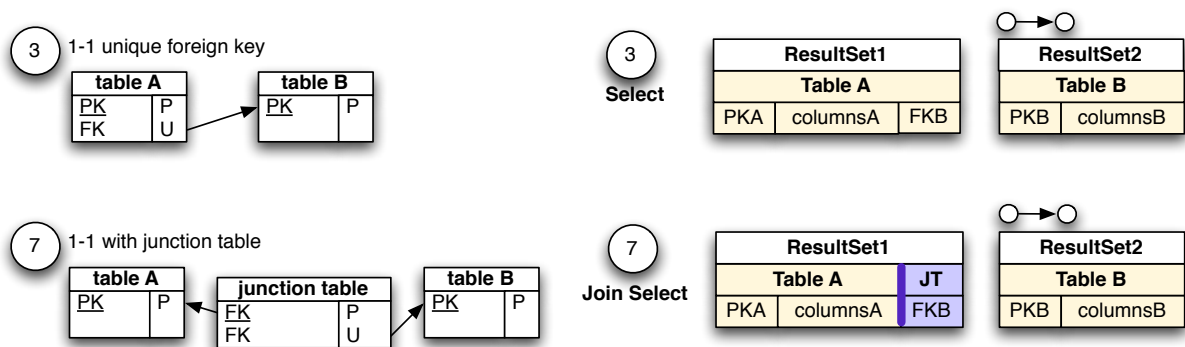


Figure 21: Query behaviour hypothesis 2

In this test we related two objects (of the same type) 30.000 times with each other, using the two mapping configurations described above. See Figure 22 and corresponding summary Table 3.

For both the results of the test with the Oracle as with the MySQL database the configurations perform almost equally (with only minor differences). As the performance is almost equal, both configurations can be chosen. Choosing configuration two however can prevent extra queries when using bidirectional relationships (see the possible extra costs of bidirectional relationships in hypothesis 11).

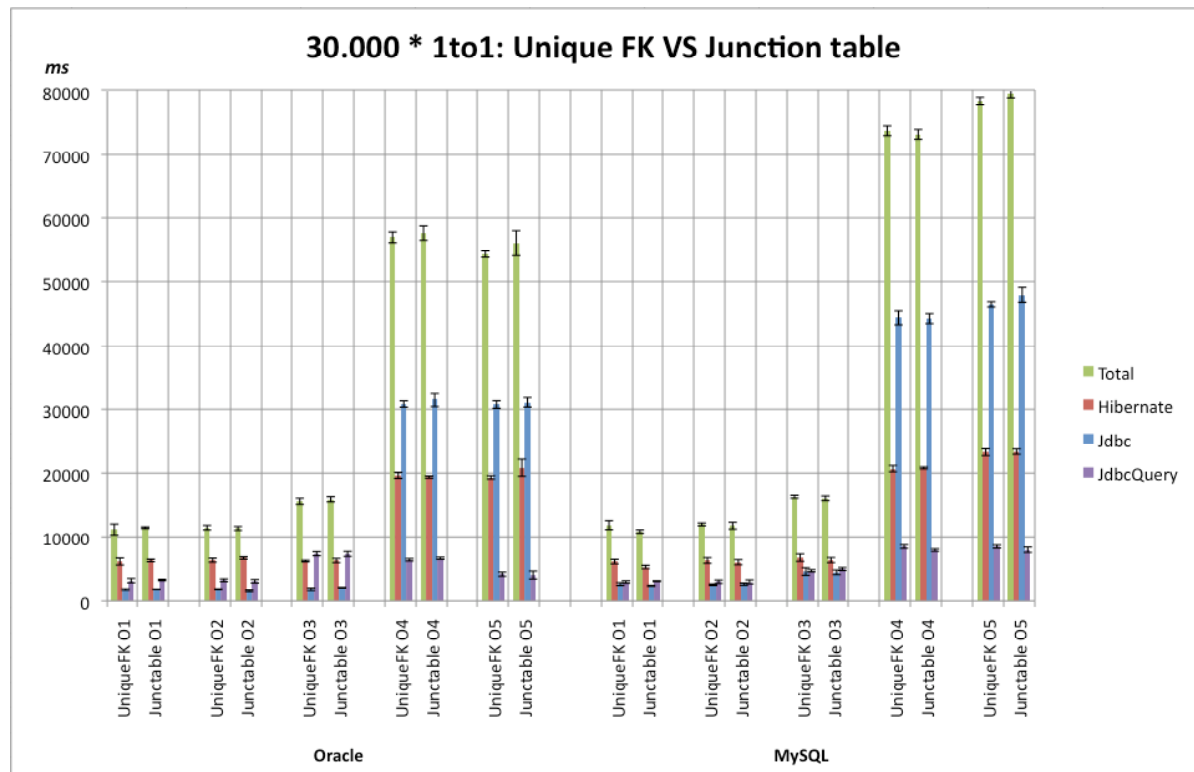


Figure 22: Performance measurements hypothesis 2

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	2,5%	7,5%	2,5%	8,9%	2,0%	8,6%	2,8%	10,4%
O2	-1,0%	3,0%	5,5%	5,0%	-16,6%	6,9%	-5,0%	8,2%
O3	2,0%	3,0%	1,2%	5,3%	14,5%	13,7%	-0,5%	5,2%
O4	1,1%	1,9%	-1,4%	2,4%	2,2%	3,5%	3,7%	3,0%
O5	3,2%	3,6%	7,8%	6,9%	1,1%	2,4%	-3,1%	14,5%
MySQL	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-8,2%	5,8%	-13,5%	5,7%	-10,2%	8,3%	4,4%	6,1%
O2	-1,7%	4,4%	-4,3%	7,0%	3,4%	6,7%	-0,6%	10,0%
O3	-1,4%	2,2%	-5,6%	8,5%	-2,8%	12,7%	6,0%	4,9%
O4	-0,8%	1,0%	0,5%	2,4%	-0,3%	2,4%	-6,4%	3,3%
O5	1,5%	0,8%	0,4%	2,3%	3,3%	2,7%	-5,9%	5,0%

Table 3: Summary performance measurements hypothesis 2

## Conclusion

Storing a relationship in a separate table does not influence the performance noticeably. In most cases the measured differences are neglectable but in two cases the results deviate a bit more (but still minimal). The use of a junction table to store the foreign keys in a one to one relationship can therefore be neglected. The use of this configuration can thus be considered when other configurations cause extra overhead.



### 7.1.2 Testing the amount of queries

- 3) **Hypothesis:** Retrieving two objects from one table will perform faster than retrieving two objects from two tables, using two select queries.

Accepted

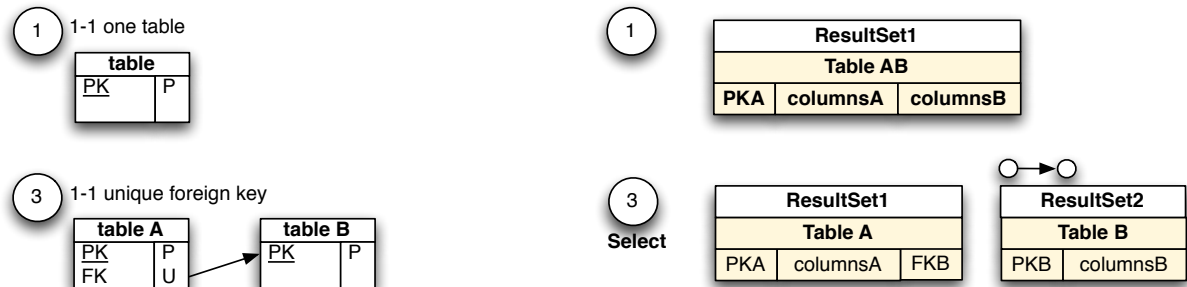


Figure 23: Query behaviour hypothesis 3

In this test we related two objects (of the same type) 30.000 times with each other, using the two mapping configuration described above. See Figure 24 and corresponding summary Table 4.

For the smaller objects the difference between configuration one and two is significantly large, while this difference becomes smaller when the object grows in size. The cause is (for the tests with both databases) the JDBC driver that drops in performance drastically, when the amount of columns increases.

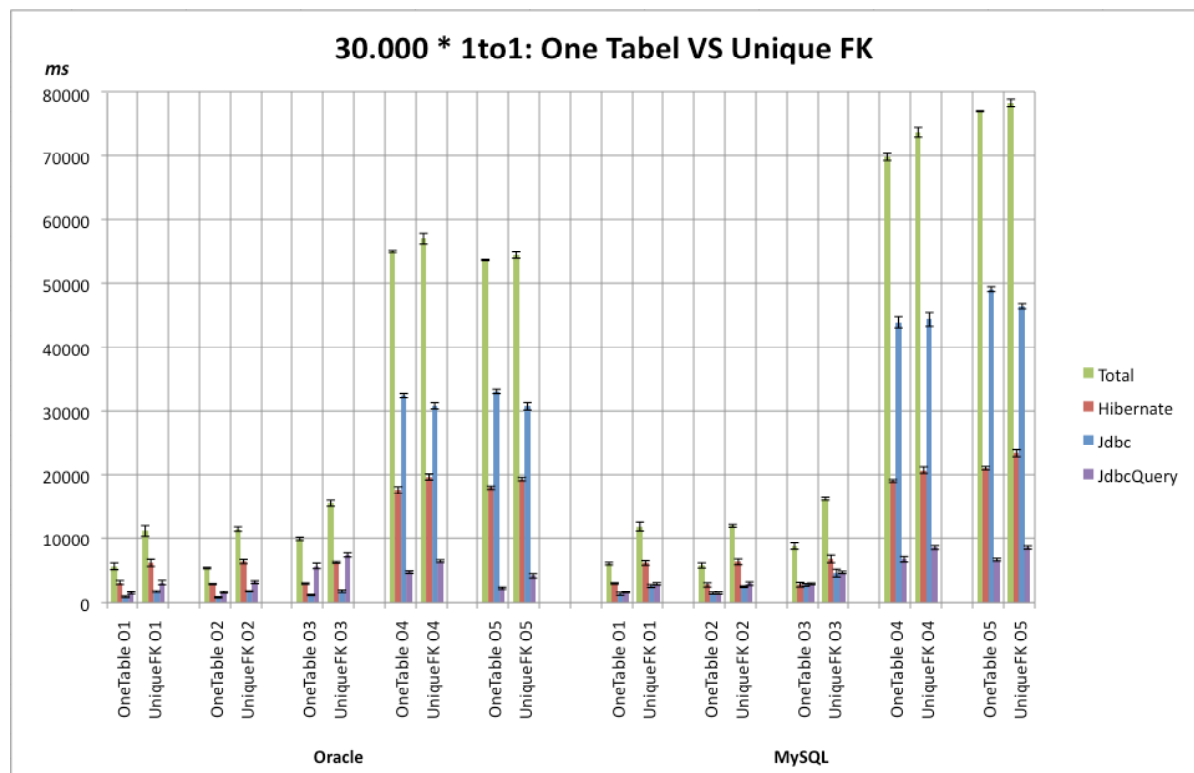


Figure 24: Performance measurements hypothesis 3

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	97,2%	14,8%	91,9%	17,0%	95,3%	16,8%	109,4%	21,7%
O2	112,8%	6,5%	114,9%	10,6%	125,5%	13,7%	102,7%	13,4%
O3	55,3%	4,7%	105,8%	4,4%	53,8%	21,1%	28,8%	7,3%
O4	3,7%	1,5%	11,6%	3,0%	-5,1%	1,5%	34,7%	4,1%
O5	1,4%	0,9%	7,4%	1,4%	-7,4%	1,8%	79,8%	13,7%
MySQL								
O1	95,5%	11,4%	101,2%	11,4%	94,9%	16,6%	85,4%	11,3%
O2	106,4%	6,7%	122,5%	15,7%	75,8%	10,7%	105,8%	18,0%
O3	84,7%	5,7%	135,2%	20,0%	62,5%	20,6%	57,4%	5,8%
O4	5,5%	1,1%	8,7%	2,6%	1,0%	2,4%	26,3%	6,1%
O5	1,8%	0,8%	10,8%	2,6%	-5,6%	0,8%	27,6%	3,7%

Table 4: Summary performance measurements hypothesis 3

## Conclusion

Retrieving two objects from one table performs faster than retrieving them from two tables with two select queries, as expected. When the size of the objects grows this difference is decreasing.

### 7.1.3 Testing all factors

- 4) **Hypothesis:** Retrieving two objects from two tables with a join query will perform faster than retrieving them using two select queries.

Accepted

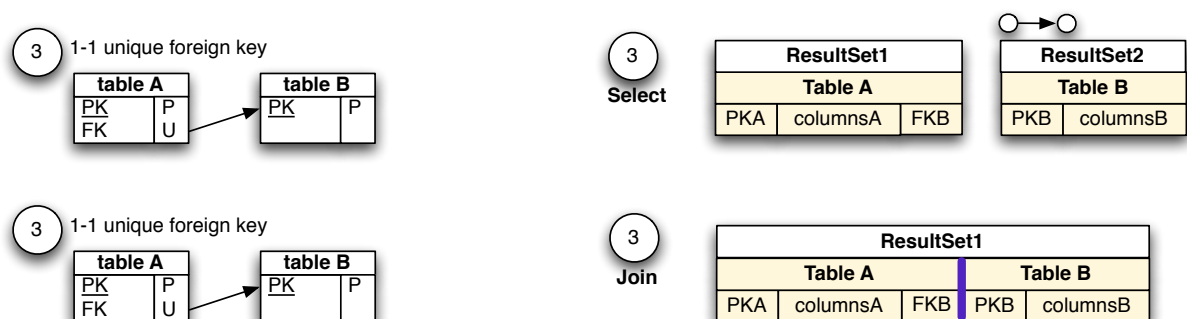


Figure 25: Query behaviour hypothesis 4

In this test we related two objects (of the same type) 30.000 times with each other, using the two mapping configuration described above. See Figure 26 and corresponding summary Table 5.

The first things you notice are the problems the oracle JDBC driver has with processing the results of a joined query with a lot of columns. In all situations the joined query performs faster than two select queries, except with large objects. With the results of the tests with MySQL the performance gain is reduced (but it stays faster) and with the results of the tests with the oracle database the performance order even reverses.

Both databases (JdbcQuery time) as well as Hibernate do perform one join query faster than two select queries.

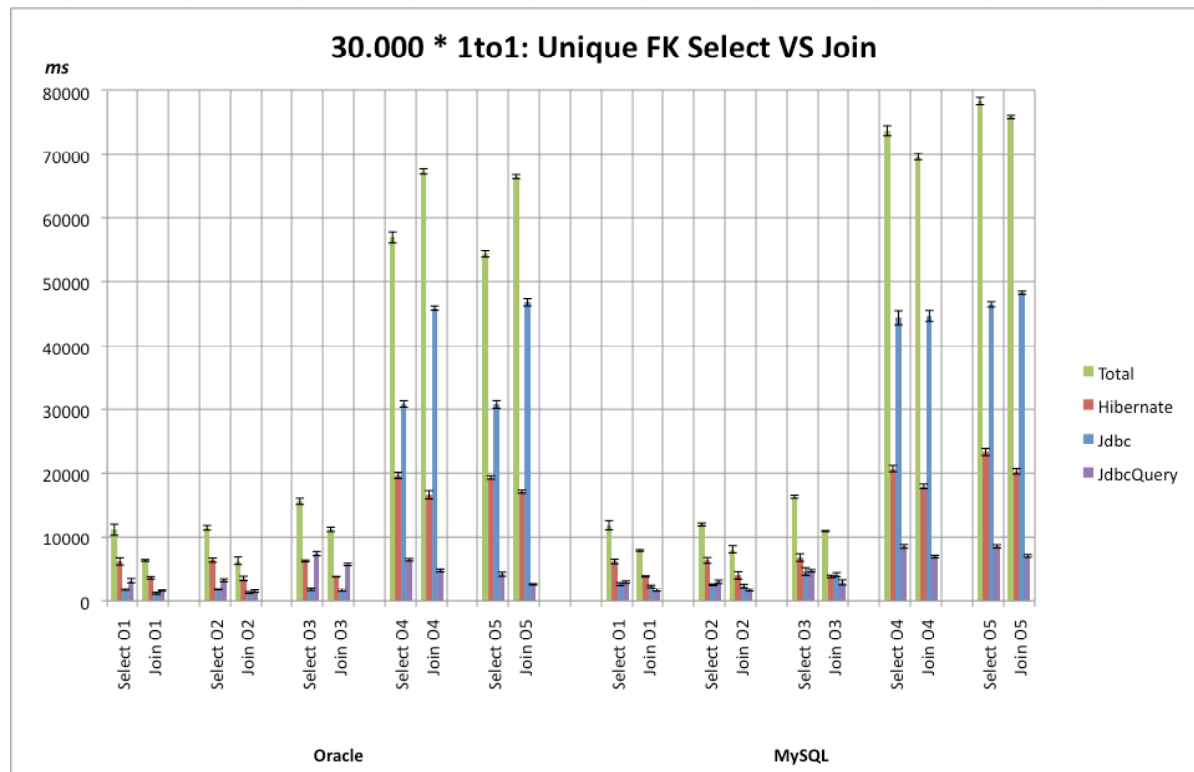


Figure 26: Performance measurements hypothesis 4

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-43,3%	7,5%	-41,0%	8,9%	-36,1%	8,6%	-51,6%	10,4%
O2	-45,1%	5,1%	-44,2%	5,3%	-30,7%	7,9%	-54,9%	6,6%
O3	-27,4%	3,0%	-38,7%	2,1%	-8,3%	13,7%	-22,5%	4,4%
O4	18,0%	1,5%	-15,5%	3,6%	48,7%	1,6%	-25,7%	3,4%
O5	22,3%	0,9%	-11,7%	1,6%	52,0%	1,9%	-37,1%	7,6%
MySQL	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-33,7%	5,8%	-37,2%	5,7%	-13,7%	8,3%	-44,4%	6,5%
O2	-32,7%	4,6%	-36,4%	9,0%	-9,6%	11,3%	-44,5%	8,7%
O3	-32,0%	1,4%	-43,0%	8,5%	-10,3%	12,7%	-37,7%	8,3%
O4	-5,5%	1,0%	-13,2%	2,4%	0,7%	2,4%	-18,6%	3,3%
O5	-3,2%	0,8%	-12,8%	2,3%	4,2%	0,9%	-17,3%	2,9%

Table 5: Summary performance measurements hypothesis 4

## Conclusion

In the large amount of the cases retrieving two objects from two tables with a joined query is faster than retrieving them by two separate select queries. When this is not true the oracle JDBC driver has to process the results of retrieving large objects with a join query. We assume that this is due to an error in the implementation of the Oracle JDBC driver and will take this into account in analyzing the rest of the results.

- 5) **Hypothesis:** Retrieving one object from one table will perform faster than retrieving two objects using a join query.

Accepted

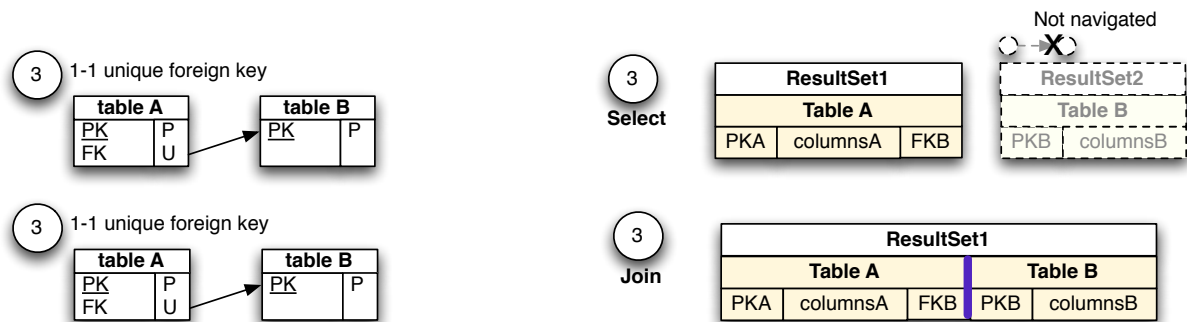


Figure 27: Query behaviour hypothesis 5

In this test we related two objects (of the same type) 30.000 times with each other, using the two mapping configuration described above. See Figure 28 and corresponding summary Table 6.

In the previous hypotheses we observed that the performance is bad whilst retrieving larger objects (especially with a joined query). Therefore, it is logical to see the great performance gains of configuration one with large objects, while with smaller objects this gain is much less.

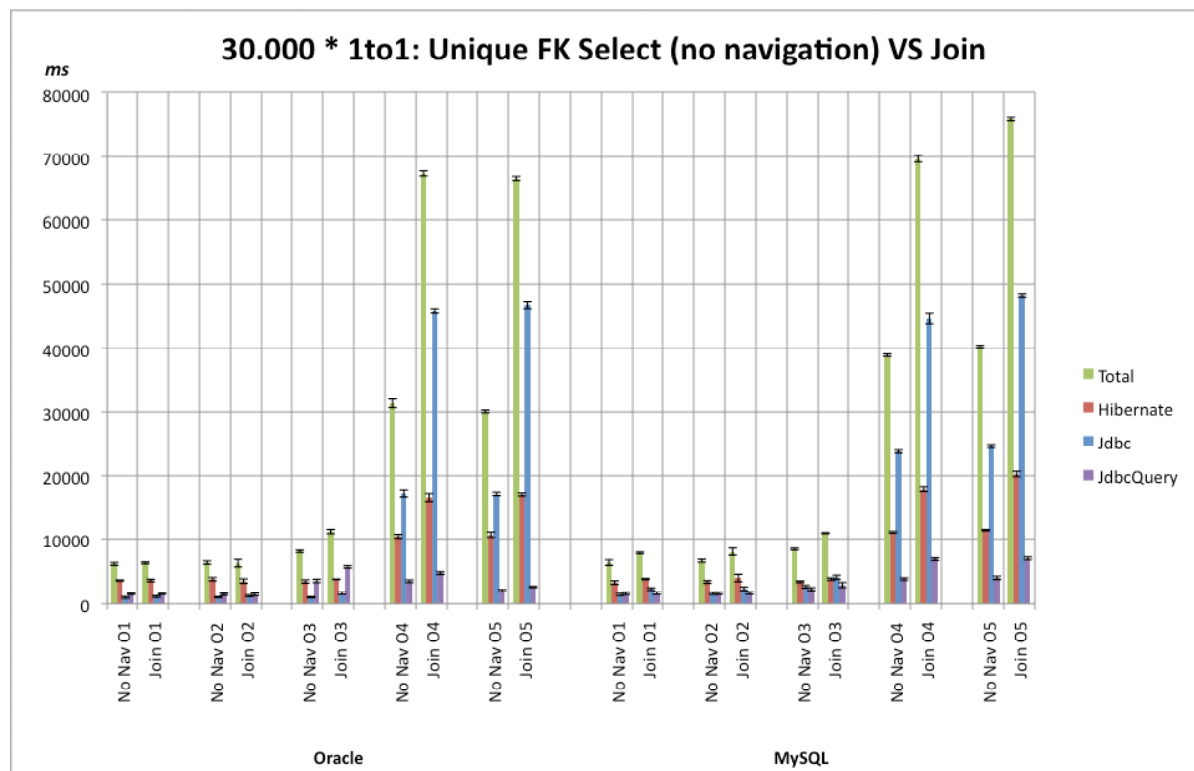


Figure 28: Performance measurements hypothesis 5

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	2,4%	3,6%	0,2%	5,2%	13,7%	16,1%	0,5%	6,3%
O2	-1,4%	9,1%	-7,6%	8,8%	21,8%	13,8%	-1,5%	13,1%
O3	38,4%	4,1%	9,5%	6,9%	63,6%	19,2%	59,4%	7,1%
O4	114,5%	2,2%	58,1%	6,7%	165,1%	3,5%	35,5%	6,2%
O5	121,1%	1,0%	58,6%	3,9%	171,3%	3,2%	28,8%	8,5%
MySQL								
O1	23,1%	6,7%	16,2%	8,2%	61,0%	13,3%	3,3%	12,0%
O2	20,7%	8,2%	17,5%	16,7%	45,5%	18,2%	3,1%	11,5%
O3	30,0%	1,7%	12,0%	5,5%	54,2%	11,4%	28,5%	17,1%
O4	79,1%	1,4%	61,2%	3,1%	87,7%	3,5%	78,3%	5,3%
O5	88,3%	0,6%	76,7%	3,6%	96,2%	1,1%	73,4%	6,3%

Table 6: Summary performance measurements hypothesis 5

## Conclusion

In most cases not retrieving the data performs better than retrieving it in a join query with the original query, in two of the ten cases this difference is neglectable. The difference increases rapidly when the objects grow in size. When using the oracle database, the difference between both configurations on small objects is not noticeable at all.

Retrieving an object either always by using a join query or only when needed can make a huge difference when working with large objects. The performance difference is especially noticeable in the JDBC driver, but as this is very small with small objects the total performance does not differentiate that much. As the objects get larger the difference on the other aspects also increases.

- 6) **Hypothesis:** Retrieving an one to many relationship (with 10 objects on the many side) will perform faster when the relationship is set to select fetching than to join fetching.

Rejected

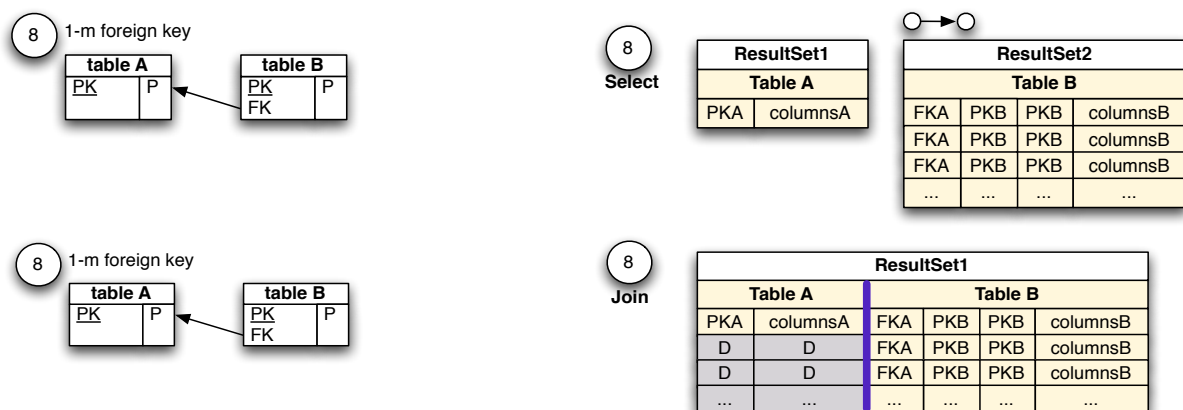


Figure 29: Query behaviour hypothesis 6

In this test we related one object with ten objects (of the same type) 3.000 times, using the two mapping configuration described above. See Figure 30 and corresponding summary Table 7.

The performance gain of configuration one compared to configuration two is minimal. Only the known performance drop of the join query is causing the oracle JDBC to perform badly. In the next

hypothesis we will compare this same situation but with a one to hundred relationship, looking what happens when more values are duplicated.

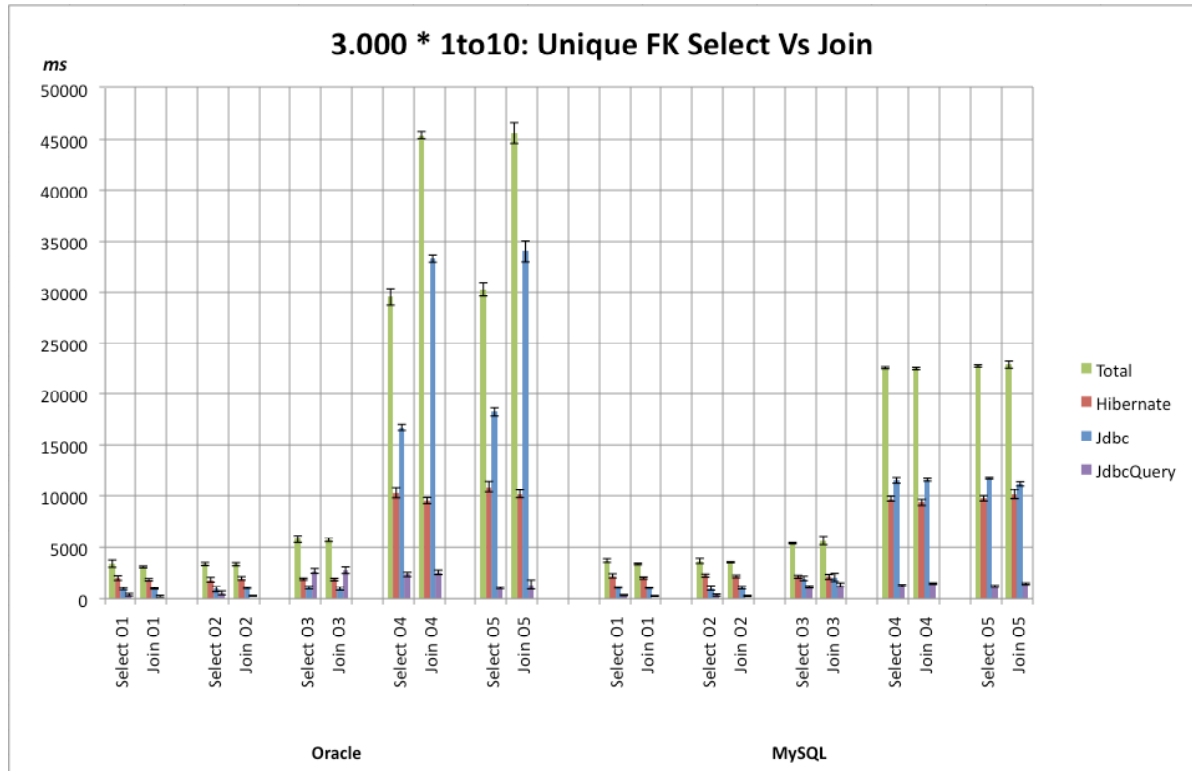


Figure 30: Performance measurements hypothesis 6

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-9,2%	10,2%	-8,1%	10,6%	1,9%	12,1%	-43,5%	32,6%
O2	-0,7%	4,5%	7,0%	11,9%	13,6%	28,5%	-51,1%	34,4%
O3	-1,5%	5,9%	-2,2%	6,1%	-10,2%	14,9%	2,6%	11,6%
O4	53,9%	2,8%	-7,6%	5,1%	98,4%	2,1%	8,4%	8,6%
O5	50,8%	3,3%	-6,5%	5,0%	86,3%	5,8%	30,3%	42,2%
MySQL								
O1	-8,9%	4,7%	-9,3%	8,4%	-3,5%	7,0%	-24,1%	17,4%
O2	-3,4%	6,6%	-3,4%	6,2%	4,4%	22,3%	-27,0%	29,7%
O3	5,4%	7,9%	-0,5%	10,5%	4,4%	19,0%	18,3%	17,8%
O4	-0,3%	0,5%	-3,7%	3,0%	0,6%	2,4%	16,7%	8,3%
O5	0,5%	1,5%	4,3%	4,6%	-4,7%	1,8%	20,7%	8,7%

Table 7: Summary performance measurements hypothesis 6

## Conclusion

In hypothesis 4 we noticed the bad performance of the oracle JDBC driver when processing large objects that are retrieved with a join. Ignoring this behaviour we observe that in a one to ten relationship the performance of a join or two separate select queries can be neglected.

7) **Hypothesis:** Retrieving an one to many relationship (with 100 objects on the many side) will perform faster when the relationship is set to select fetching than to join fetching.

Rejected

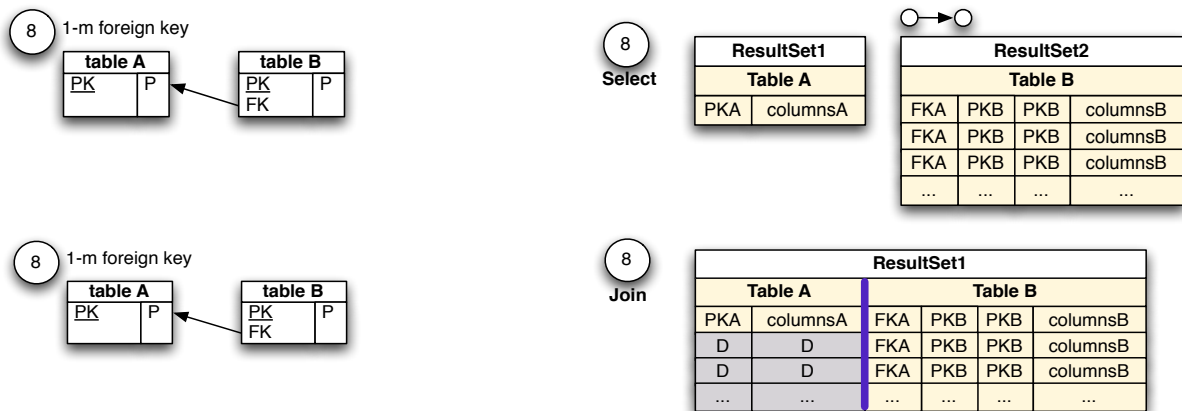


Figure 31: Query behaviour hypothesis 7

In this test we related one object with hundred objects (of the same type) 300 times, using the two mapping configuration described above. See Figure 32 and corresponding summary Table 8.

In none of the situations the joined query is significantly faster than two select queries, in two cases it is slower but these are ignored (see hypothesis 4). Therefore, duplicating values does not influence the performance with these amount of objects. This can possibly be the cause of optimization techniques in the JDBC and database.

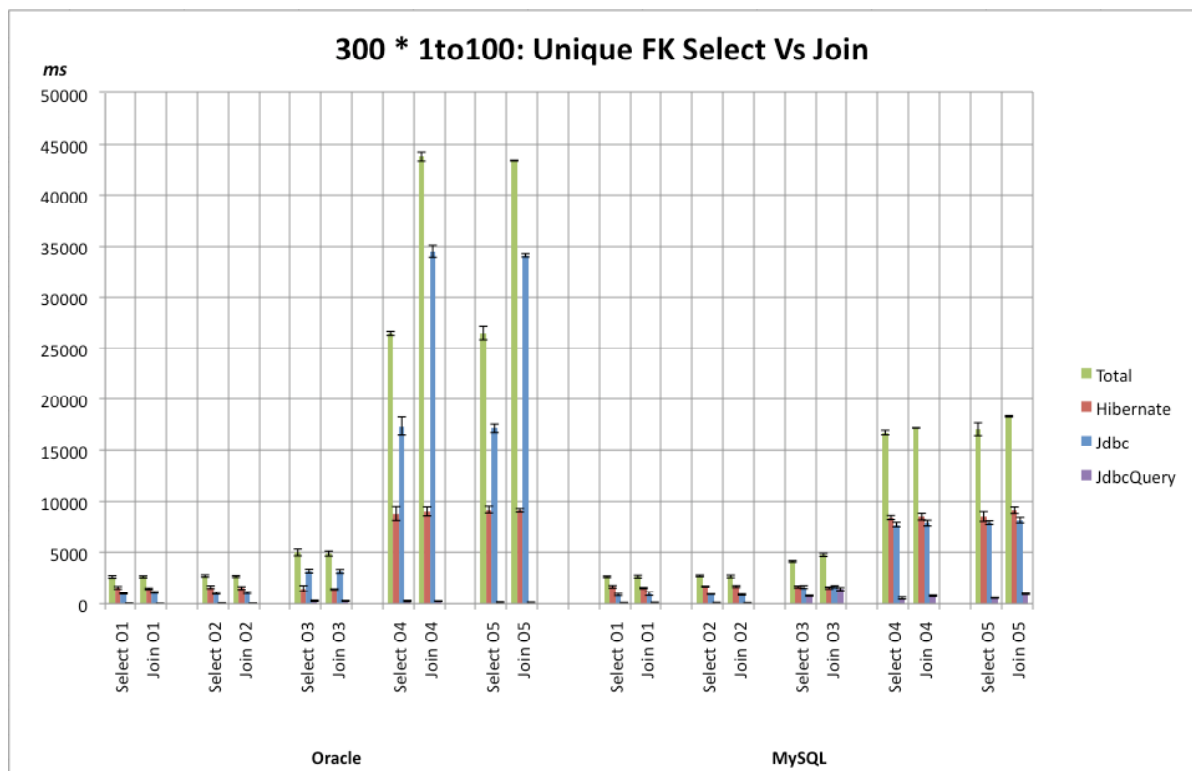


Figure 32: Performance measurements hypothesis 7

<b>Oracle</b>	<b>Total</b>	<i>std</i>	<b>Hibernate</b>	<i>std</i>	<b>Jdbc</b>	<i>std</i>	<b>JdbcQuery</b>	<i>std</i>
<b>O1</b>	0,3%	4,6%	-6,6%	8,1%	12,9%	8,4%	-40,6%	83,7%
<b>O2</b>	-1,7%	4,0%	-5,5%	8,8%	6,1%	8,9%	-33,6%	59,9%
<b>O3</b>	-2,2%	6,5%	-4,6%	18,9%	-0,9%	5,6%	-4,2%	22,3%
<b>O4</b>	65,5%	1,8%	2,4%	7,6%	98,4%	5,0%	-1,7%	29,1%
<b>O5</b>	63,6%	2,4%	-0,8%	3,5%	98,6%	2,4%	6,4%	39,6%
<b>MySQL</b>								
<b>O1</b>	0,6%	5,0%	-7,8%	7,0%	11,4%	20,7%	71,5%	80,9%
<b>O2</b>	-2,0%	4,7%	0,0%	5,4%	-5,0%	7,7%	-9,3%	48,3%
<b>O3</b>	15,3%	3,5%	-6,2%	5,6%	2,8%	8,8%	88,5%	19,6%
<b>O4</b>	2,7%	1,3%	0,9%	3,9%	1,8%	3,6%	41,2%	18,9%
<b>O5</b>	7,2%	3,6%	7,1%	5,6%	2,9%	3,3%	69,7%	11,6%

**Table 8: Summary performance measurements hypothesis 7**

## Conclusion

In hypothesis 6 we compared the join query with two select queries to a one to ten relationship. In this hypothesis the same test was performed but with a one to hundred relationship. The results stay neglectable between the two methods, except when the MySQL database is used for retrieving small objects with a large string. As this is the only time we observe this behaviour of object 3 and only in one of the ten tests the performance differences is not neglectable, we consider the performance differences between the techniques to be neglectable.



- 8) **Hypothesis:** Join fetching the junction table with the many side will perform faster than select fetching each object from the many side, with an amount of 10 objects at the many side.

Accepted

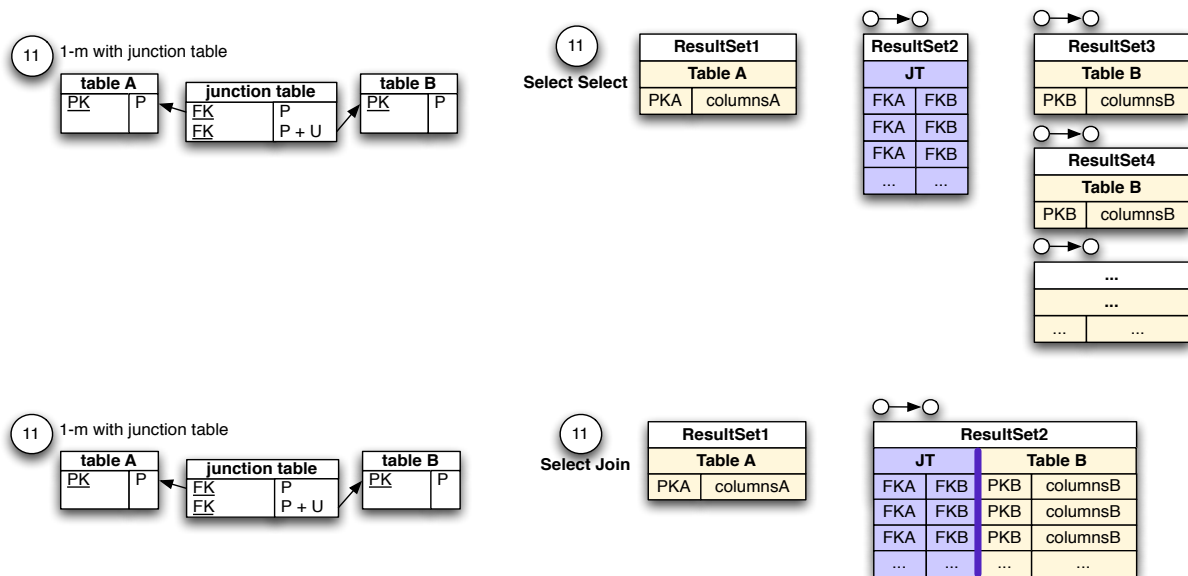


Figure 33: Query behaviour hypothesis 8

In this test we related one object with ten objects (of the same type) 3.000 times, using the two mapping configuration described above. See Figure 34 and corresponding summary Table 9.

In all situations “configuration two” performs significantly faster than “configuration one”, even with the bad performance of the oracle JDBC driver (when joining large objects).

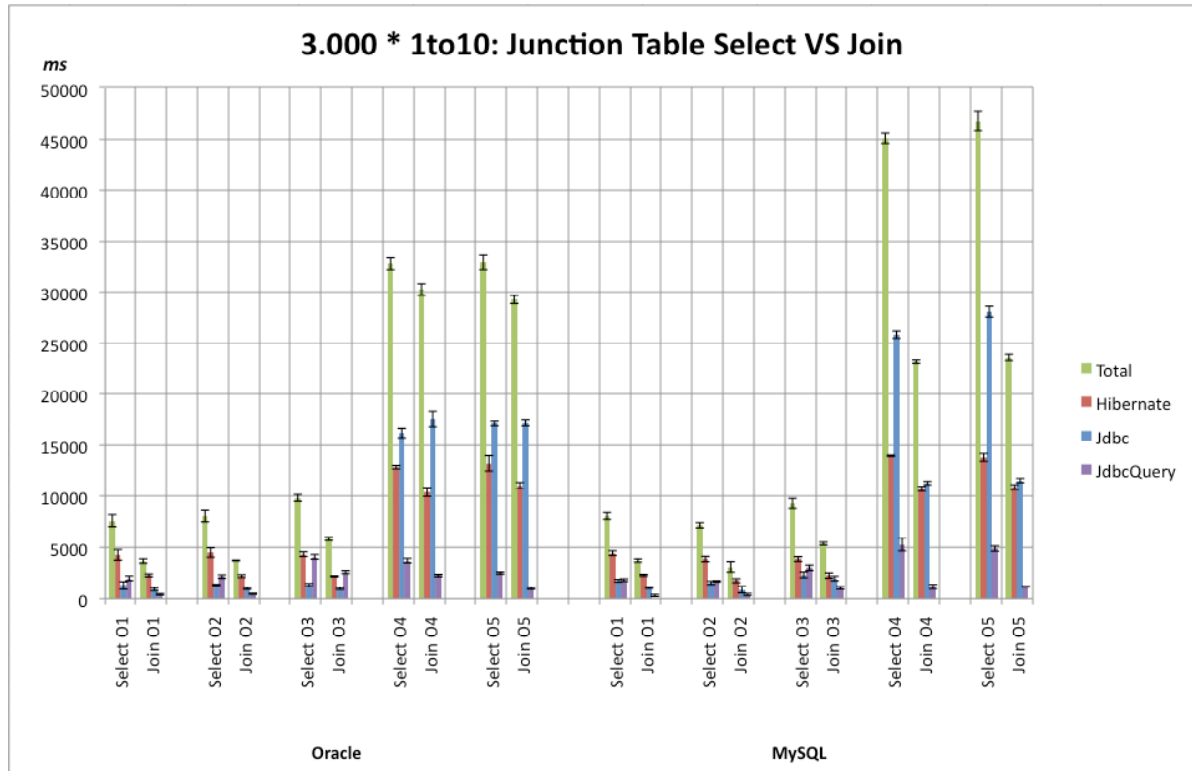


Figure 34: Performance measurements hypothesis 8

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-51,7%	7,6%	-46,0%	11,9%	-28,4%	28,2%	-78,8%	11,3%
O2	-53,7%	6,9%	-50,4%	10,2%	-26,1%	6,5%	-77,8%	7,5%
O3	-40,4%	3,4%	-49,0%	5,1%	-26,8%	9,9%	-35,8%	5,5%
O4	-7,7%	1,8%	-19,1%	3,3%	8,3%	4,5%	-38,5%	5,9%
O5	-11,1%	2,2%	-16,7%	5,8%	0,3%	1,6%	-59,6%	3,9%
MySQL	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-53,8%	4,1%	-48,0%	5,1%	-38,2%	7,1%	-82,9%	6,7%
O2	-56,5%	7,1%	-53,7%	6,6%	-41,9%	21,0%	-76,0%	6,2%
O3	-42,0%	5,2%	-40,9%	6,6%	-15,1%	11,5%	-64,4%	8,5%
O4	-48,6%	1,1%	-23,3%	1,6%	-56,4%	1,4%	-77,4%	11,8%
O5	-49,5%	2,0%	-21,3%	2,8%	-58,9%	1,9%	-75,5%	5,1%

Table 9: Summary performance measurements hypothesis 8

## Conclusion

Using a join at the right places within a one to many relationship that uses a junction table to store the relationship can spare a lot of SQL queries. Preventing queries will perform faster in all situations. Joining the many side with the junction table prevents a single query for each ID stored in the junction table.

- 9) **Hypothesis:** Join fetching the junction table with the many side will perform faster than select fetching each object from the many side, with an amount of 100 objects at the many side.

Accepted

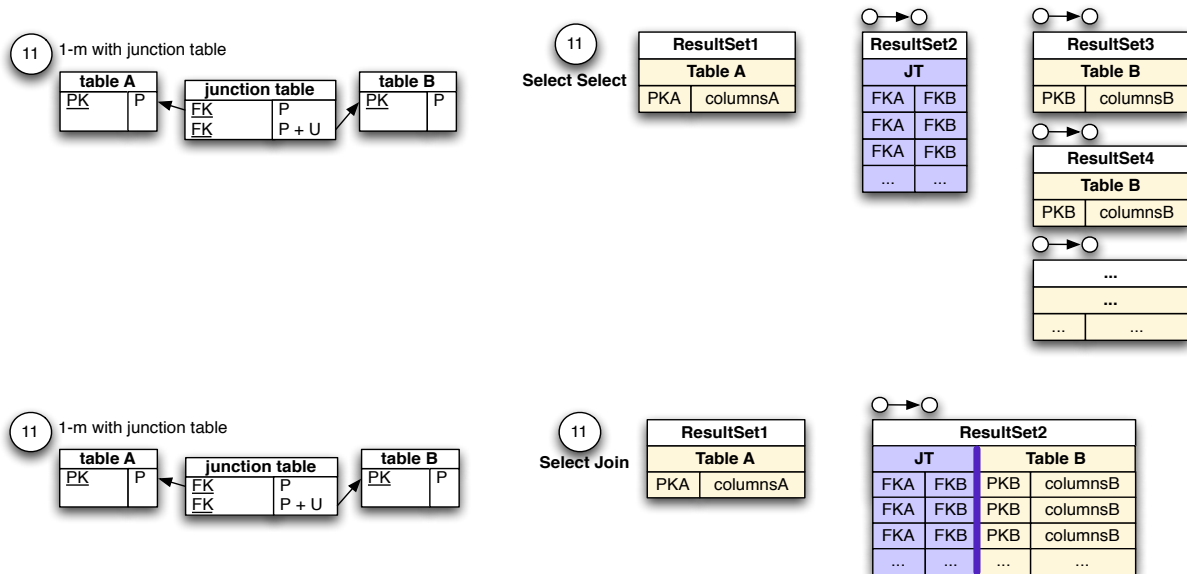


Figure 35: Query behaviour hypothesis 9

In this test we related one object with hundred objects (of the same type) 300 times, using the two mapping configuration described above. See Figure 36 and corresponding summary Table 10. This test is to demonstrate the performance loss when increasing the amount of related objects by ten. When this amount is further increased, this performance drop will even be more drastically.

Increasing the amount of related objects makes configuration two perform even faster compared to configuration one.

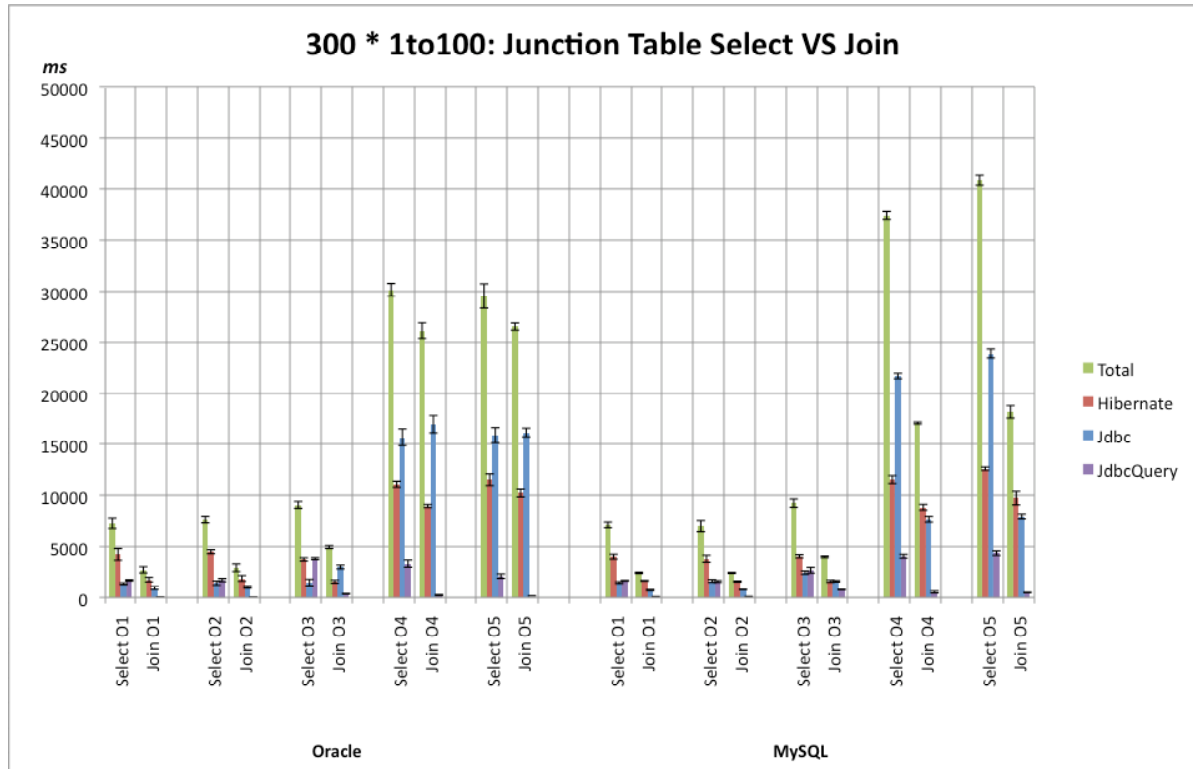


Figure 36: Performance measurements hypothesis 9

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	-62,5%	6,7%	-58,0%	13,4%	-31,8%	12,1%	-98,2%	4,0%
O2	-61,6%	4,9%	-57,8%	6,3%	-28,7%	14,3%	-98,9%	8,8%
O3	-45,3%	3,6%	-57,9%	4,2%	103,5%	22,1%	-90,7%	2,4%
O4	-13,3%	2,5%	-19,4%	2,6%	8,2%	5,4%	-93,5%	10,5%
O5	-10,1%	4,0%	-11,3%	4,8%	1,8%	4,8%	-93,1%	10,0%
MySQL								
O1	-65,9%	3,7%	-58,6%	5,9%	-52,1%	6,7%	-95,6%	2,5%
O2	-65,4%	7,5%	-58,4%	8,9%	-53,1%	8,5%	-94,9%	6,2%
O3	-56,7%	4,4%	-60,2%	3,8%	-34,4%	6,5%	-71,8%	10,3%
O4	-54,3%	1,0%	-23,6%	3,3%	-64,5%	1,2%	-87,0%	4,4%
O5	-55,5%	1,5%	-22,9%	5,2%	-66,6%	1,8%	-89,0%	5,1%

Table 10: Summary performance measurements hypothesis 9

## Conclusion

Increasing the amount of related objects of the test in hypothesis 8 will confirm that with more relationship the performance differences drastically grow even more.

10) **Hypothesis:** Storing the foreign key within the objects table will perform faster than storing it in a junction table, with a one to many relationship and an amount of ten objects at the many side.

Rejected

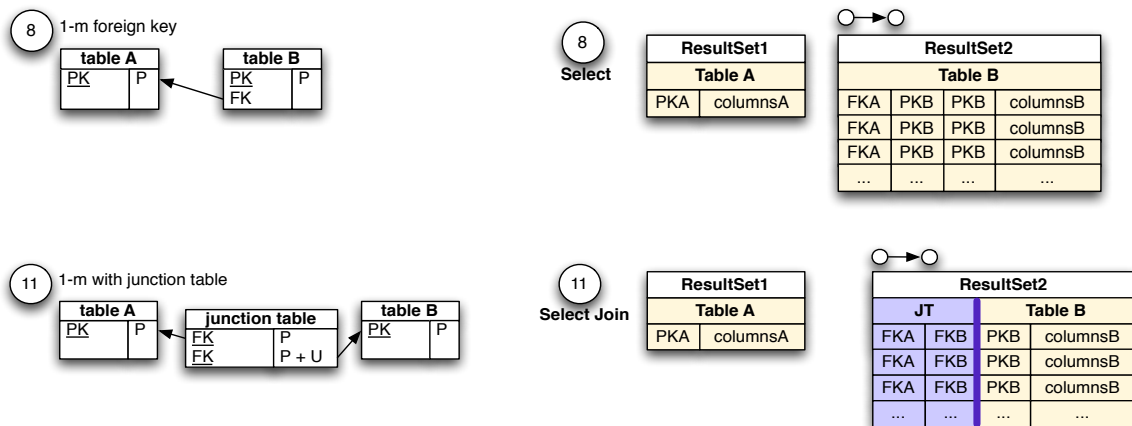


Figure 37: Query behaviour hypothesis 10

In this test we related one object with ten objects (of the same type) 3.000 times, using the two mapping configuration described above. See Figure 38 and corresponding summary Table 11.

For both databases the configurations perform almost equally. Joining a junction table with an ordinary object table does not cause great performance drops in the oracle JDBC driver.

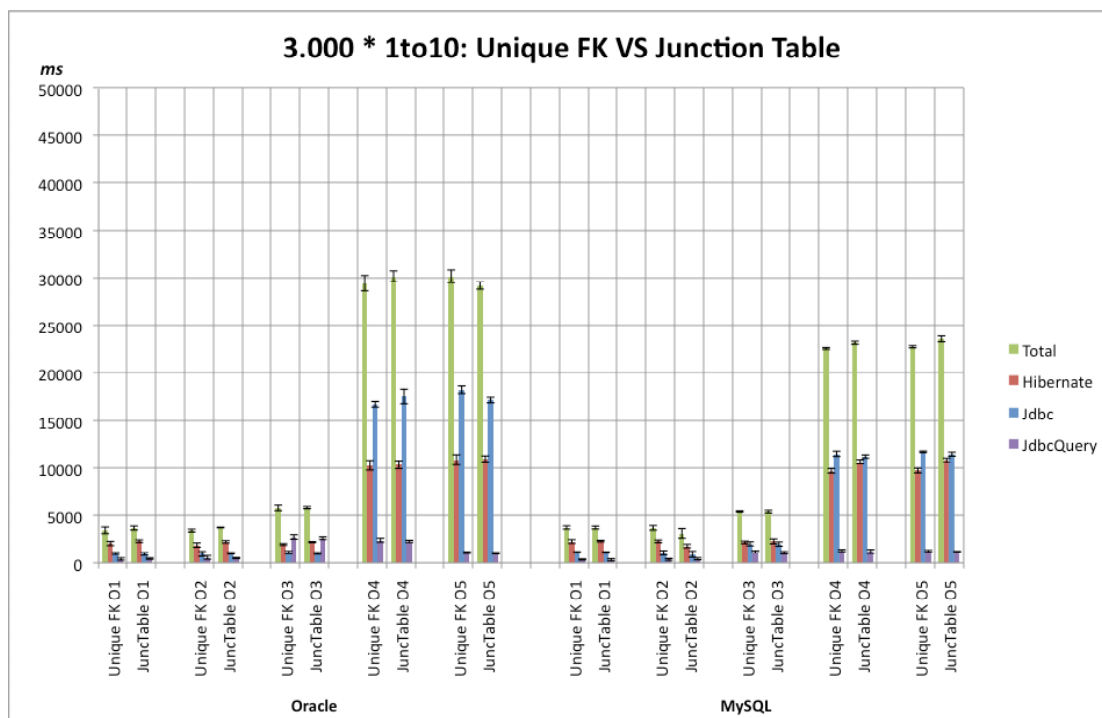


Figure 38: Performance measurements hypothesis 10

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	7,0%	10,2%	11,7%	10,6%	-4,4%	16,1%	11,1%	32,6%
O2	9,5%	4,4%	17,5%	11,9%	7,0%	28,5%	-13,3%	34,4%
O3	0,5%	5,9%	13,0%	4,9%	-9,5%	13,1%	-4,4%	8,4%
O4	2,5%	2,8%	0,6%	5,1%	4,7%	4,4%	-5,0%	8,4%
O5	-3,3%	2,3%	0,9%	5,0%	-5,7%	2,2%	-5,4%	8,9%
MySQL								
O1	-0,2%	4,7%	1,5%	8,4%	-1,8%	7,0%	-6,9%	30,4%
O2	-15,6%	13,7%	-21,8%	8,2%	-13,7%	31,2%	19,1%	30,8%
O3	-0,2%	2,6%	4,9%	11,7%	-1,4%	11,0%	-7,7%	11,5%
O4	2,6%	0,7%	10,2%	2,4%	-2,5%	2,4%	-8,4%	16,2%
O5	3,6%	1,3%	11,5%	2,6%	-2,1%	1,8%	-3,8%	8,7%

Table 11: Summary performance measurements hypothesis 10

## Conclusion

In hypothesis 2 we tested if a join with the junction table could be neglected for a one to one relationship and this proved to be the case. In hypothesis 10 we confirmed that this is also the case when joining a one to ten relationship. In some situations the one will perform slightly faster than the other, but in most cases the measurements overlap each other.

11) **Hypothesis:** Using a mapping configuration that does not introduce overhead when configuring a bidirectional relationship will perform faster than using a mapping configuration that does.

Accepted

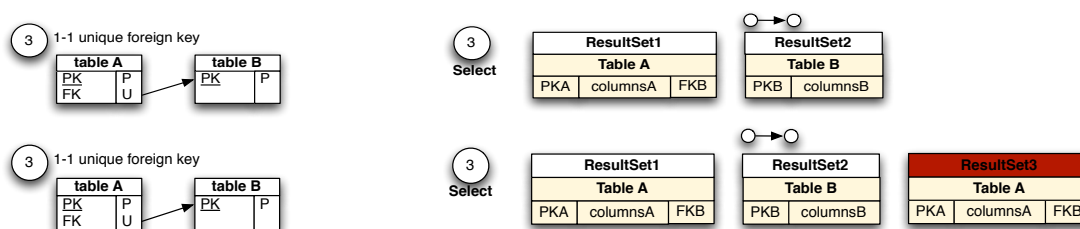


Figure 39: Query behaviour hypothesis 11

In this test we related two objects (of the same type) 30.000 times with each other, using the two mapping configuration described above. See Figure 40 and corresponding summary Table 12.

In none of the situations the performance difference reaches 50%, meaning that the extra query (that retrieves an already retrieved object again) is performed in less time and possibly optimized. With larger objects the difference drops even further.

Although it runs a bit optimized, the overhead still deteriorates the performance. Choosing another mapping configuration in this case can perform better (like the configuration were the relationship is stored in a junction table).

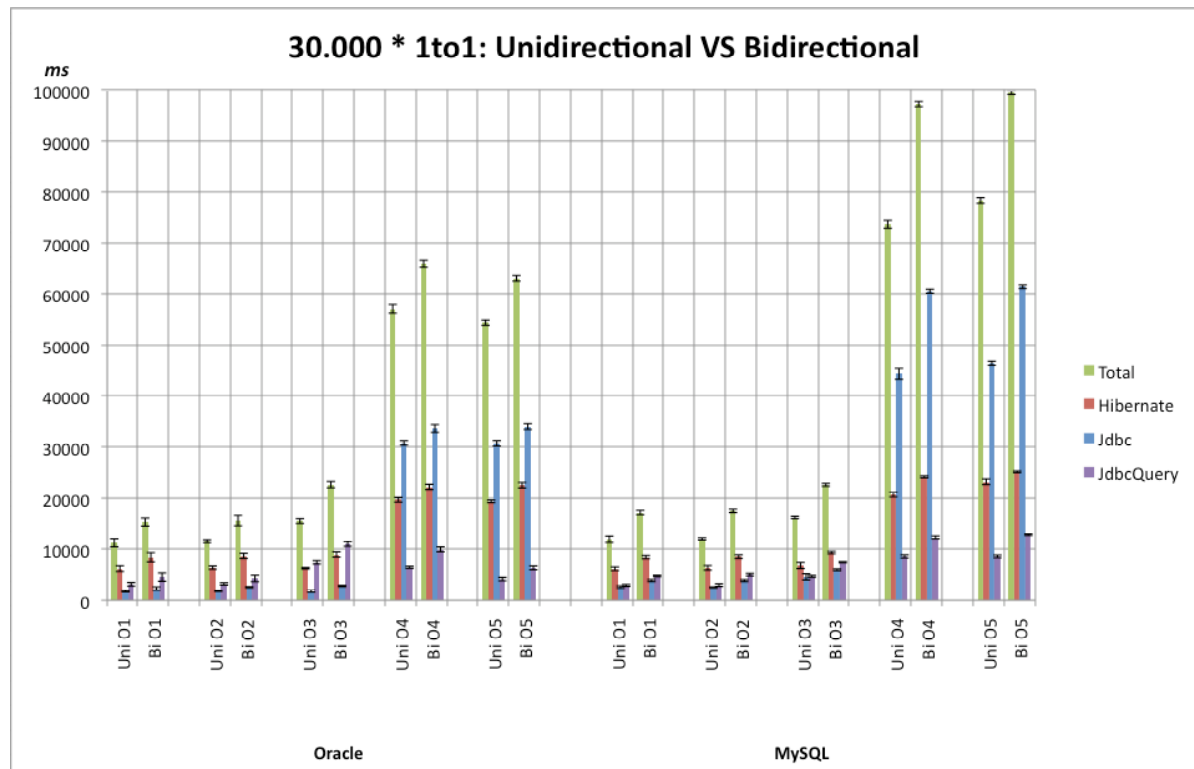


Figure 40: Performance measurements hypothesis 11

Oracle	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	36,4%	7,5%	34,8%	14,1%	31,5%	20,1%	42,4%	24,5%
O2	35,4%	8,8%	34,6%	7,6%	44,2%	6,7%	32,1%	19,0%
O3	46,4%	3,9%	41,8%	8,0%	58,2%	13,7%	47,4%	6,3%
O4	15,6%	1,5%	13,3%	2,6%	9,3%	2,4%	52,8%	7,4%
O5	16,0%	1,0%	16,9%	2,7%	10,7%	1,9%	50,5%	8,7%
MySQL	Total	std	Hibernate	std	Jdbc	std	JdbcQuery	std
O1	44,3%	5,8%	35,8%	5,7%	46,3%	8,3%	59,9%	6,1%
O2	45,5%	2,8%	33,3%	7,0%	51,4%	7,5%	66,2%	8,7%
O3	39,9%	1,7%	36,1%	8,5%	28,6%	12,7%	56,3%	3,7%
O4	32,0%	1,0%	16,8%	2,4%	36,6%	2,4%	44,7%	3,4%
O5	27,2%	0,8%	8,1%	2,3%	32,4%	0,9%	50,7%	2,9%

Table 12: Summary performance measurements hypothesis 11

## Conclusion

Choosing certain configurations for bidirectional relationships can cause an overhead that will drastically deteriorate the performance. This overhead is retrieving previously retrieved data again, slightly optimized but still takes a lot of time to execute. Choosing a mapping configuration that will not cause this overhead can be recommended in these cases.

## 7.2 Threats to validity

Every experimental research setup has certain threats to validity. In this chapter we discuss several of them, categorising them under internal and external threats.

### 7.2.1 Internal

#### Using the Get method of the API

Before relationships can be retrieved, a starting object is needed. In order to retrieve this starting object Hibernate supports several methods (in its API). In our research we only used the Get method, which queries an object by its primary key but Hibernate also supports several other querying methods. As we measure both the querying of the first object as querying the relationships together, the total performance and the differences expressed in percentages (of this total) will change when using these different methods. The relation querying will however not change and the position (faster or slower) compared to each other will not change as well and will therefore not endanger the results or conclusions of this research.

#### Possible bug in JDBC driver

We also discovered strange behaviour when joining large objects when using the oracle JDBC driver. It is therefore not entirely sure if there are more effects on the performance that went unnoticed. We also noticed strange behaviour when iterating through a large ResultSet; it takes a lot of time for both JDBC drivers to perform this action (compared to the time spent in Hibernate and the database). It might be possible that this will be solved in future releases but as these were the latest drivers available at the time and are also used in production environments, using these drivers makes the results more realistic.

#### Experimental setup

Besides the type of mapping configurations and fetching strategies, there are several other factors that influence the performance. In our research we choose a fixed value for these remaining factors. This means that we execute the tests with only five types of objects, two database implementations, a fixed amount of data in the database (max 30.000 objects in a table), the foreign keys are fixed (one data type stored in one column) and composite keys or different types are not taken into account (as the relation highly depends on these foreign keys we can imagine that the relation querying performance will also depend on this factor). What influences different choices on these factors will have is unknown. Nevertheless, this research will give a clear indication on how Hibernate behaves in this particular setting. Changing this setting will change the measured values, but the chance it will change the fastest configuration (of a particular comparison) is minimal.

The object model of our tests consists of only one relationship and it is uncertain whether the measured performance remains the same when increasing the amount of relationships in this object model. Looking at the query behaviour of Hibernate it is likely that it will, as each relationship is treated as a separate chunk and one chunk will not influence the other. The chance that a mapping configuration will therefore perform in a different way in an object model with one relationship (than with more relationships) will be very small. We also do not try to calculate the total performance; we just try to find the fastest mapping configuration, if the performance of a single query increases when the object model is increased, it does not matter as long as they all increase equally.

There is one exception on this threat, joining several queries into one query cannot be separated in different chunks. It is unclear what will happen with the performance when applying this method, we therefore recommended researching this in future work.



### **7.2.2 External**

#### **Scoping the mapping configurations**

One of the largest threats is the incompleteness of our research due to scoping: we left out some type of mapping configurations (like inheritance relationships and the lazy attribute fetching) and fetching strategies (Subselect and Batch fetching). Certain situations or exceptions are therefore not treated and information is missing to create the fastest configuration for ever situation, but with the information available a decent start can be made and poor performance can be prevented.

#### **Simple non-realistic environment**

Our measurements indicate the behaviour of a very simple non-realistic environment. There is no concurrency, no caching, no production machines, a private network (without other machines attached to it) and no database optimization. This will make the results less realistic, but will help executing different tests under the same circumstances. The results will therefore be more constant and will give a clear indication how the configurations perform in an idealistic situation. The results will however suffice to determine what configuration will prevent poor performance.

## 8 Conclusion

We have created a theory about the relation querying behaviour of Hibernate and compared some key differences between the executed SQL statements (in this behaviour) by executing several performance measurements, answering the research question: What is the effect on the performance of Hibernate querying objects when using different mapping configurations?

We can conclude that changing the mapping configuration of a persistence tool as Hibernate can have great influence on the performance. There are several ways an object relationship can be mapped to tables and several ways how Hibernate can collect it (the fetching strategy), configurable in this mapping configuration. By choosing the right table representation and fetching strategy easy performance gains can be achieved.

We also like to notice that before benchmarking several of these persistence tools (like in [1, 15]) this should be realised. Executing these kinds of benchmarks without experience or prior research to these configurations will have no added value.

In the rest of this chapter we will discuss the rejected hypotheses and give a recommendation for configuring a good performing object model.

### 8.1 Rejected Hypotheses

For the performance measurements we constructed eleven hypotheses, of which four were rejected. This indicates possible falsehoods in the rules of thumb widely available. We like to summarize the hypotheses that were rejected (some hypotheses reinforce each other and are therefore written together).

- *Storing a foreign key within the object table will perform faster than storing it in a junction table:* this is not true. We also notice that joining smaller objects will have less negative effect on the performance than joining larger objects. This is also confirmed when using a one to many relationship;
- *Retrieving an one to many relationship (with 10 objects on the many side) will perform faster when the relationship is set to select fetching than to join fetching:* this is also not true, duplicating values will have no noticeable effect on the performance. This is also confirmed with 100 objects on the many side.

### 8.2 Recommendations.

In the next part we will give our recommendations on setting up a good performing mapping configuration for Hibernate.

The Hibernate reference documentation advises to start with setting each relationship to select fetching (with lazy collection enabled) and when bad performance is encountered several changes can be made. We advice to use the results of this research to set up a good performing starting configuration.

With the described behaviour in chapter 5, the executed SQL statements of different mapping configurations for a particular object relationship can be understood. In Appendix B: Relation querying behaviour we worked out the query behaviour of the different object relationships with this described behaviour. These can then be compared to understand the behavioural differences. Supplemented with the performance measurements in chapter 7, a good performing configuration for a particular relationship can be chosen.

We do also advice (as advised in the Hibernate reference documentation) to set each select fetched relationship to lazy collection, as this will prevent unnecessary queries. To achieve even greater performance the description of the optimization techniques in [22] will help.

## 9 Future work

In this research we set up a theory for a good performing relation querying configuration in Hibernate. This configuration can be further improved by investigating the Subselect and Batch fetching strategies, inheritance relationships, joining multiple relationships, other ORM actions (insert, update and delete), the caching optimization techniques and by how the configurations will behave in more sophisticated situations. For more information see chapter 7.2 Threats to validity.

Also the causes of certain differences in the results of this research are not investigated. To further improve Hibernate (or the JDBC drivers) a study can be performed on this subject.

Besides the relation querying, also the object querying methods of the API can be further investigated. Hibernate supports four ways to query object(s): the get method, load method, Criteria objects (by restrictions and by example) and HQL (Hibernate Query Language). Before setting up our benchmark we tested these different querying methods and we found that the HQL and criteria objects performed equally, as well did the load and the get method. What we did find is that the criteria objects performed 75 times slower than the load method did. This was while using the `.uniqueResult()` method to perform the actual querying.

Both HQL and Criteria objects support several of these methods to execute the query: `.iterate()`, `.list()`, `.uniqueResult()`, `.scroll()` that each perform differently and will have a different effect on the performance. To create the optimal querying configuration and usage, these methods and the difference between the load and criteria objects can be further investigated and maybe improved.

There are also several specialized optimization techniques that are well described in the reference documentation, but of which the effect on the performance is not tested: batch insert and update, flushing objects (due to the session cache), bypassing the session cache in total, executing direct SQL and using HQL to execute delete or update statements.

We also encountered some irregularities in the JDBC driver of both Oracle and of MySQL. Further research on these drivers can improve performance.

Finally, using the knowledge gained with this research benchmarks comparing Hibernate with other persistence tools can be improved as well.

## 10 Acknowledgements

I would like to thank all supervisors guiding this research at Logica and the UvA. At Logica I would like to thank Huub van Thienen, Co Kooijman, Ruud Rietveld, Henk Laman, Charles Liefting and in special Ingrid van Zaanen (for her time spent in process management). At the UvA I would like to thank Hans Dekkers for guiding the research at start and middle phases and Jurgen Vinju for the full on guidance at the end of this research.

Also I would like to thank everybody that contributed to this research in consults, interviews and reviews. In special Jan Willem Bunt and Marcel Cullens for their knowledge on Hibernate database access layers and databases.

Further I would like to thank my Family for putting up with me this whole year.

And last but not least I would like to thank my project group with whom I fulfilled the rest of the Master Software Engineering for the knowledge exchange and the awesome time together: Michel de Graaf, Alex Hartog and Lars de Ridder.

## 11 Bibliography

- [1] Zyl, P. V., Kourie, D. G., and Boake, A. (2006). Comparing the performance of object databases and ORM tools. ACM International Conference Proceeding Series, 204.
- [2] van Zyl, P., Kourie, D. G., Coetzee, L., and Boake, A. (2009). The influence of optimizations on the performance of an object relational mapping tool. In SAICSIT '09: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, New York, NY, USA, 2009 (pp. 150--159). ACM.
- [3] Keller, W. (1998). Object/Relational Access Layers: A Roadmap, Missing Links and More Patterns. Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing. <http://www.objectarchitects.de/>
- [4] Keller, W. (1997). Mapping Objects to Tables, A Pattern Language. . <http://www.objectarchitects.de>
- [5] Keller, W. and Coldewey, J. (1996). Relational Database Access Layers: A Pattern Language. . <http://www.objectarchitects.de/>
- [6] Senior, R., Klotz, T., and Majure, J. (22 Apr 2008). Patterns of persistence, Part 1: Strategies and best practices for modern ORM tools. Retrieved 15 Dec 2009 from <http://www.ibm.com/developerworks/java/library/j-pop1/>.
- [7] Miller, J. (April 2009). Patterns in Practice: Persistence Patterns. Retrieved 15 Dec 2009 from <http://msdn.microsoft.com/en-us/magazine/dd569757.aspx>.
- [8] Lodhi, Fakhar; Ghazali, M. A. (2007). Design of a simple and effective object-to-relational mapping technique. In SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, New York, NY, USA, 2007 (pp. 1445--1449). ACM.
- [9] Ambler, S. W. (). Mapping Objects to Relational Databases: O/R Mapping In Detail. Retrieved 15 Dec 2009 from <http://www.agiledata.org/essays/mappingObjects.html>.
- [10] Ibrahim, A. and Cook, W. R. (2006). Automatic Prefetching by Traversal Profiling in Object Persistence Architectures. ECOOP 2006 ñ Object-Oriented Programming, 4067/2006, 50-73.
- [11] Han, W.-S., Moon, Y.-S., and Whang, K.-Y. (2003). PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs. Information Sciences, 152, 47 - 61. <http://www.sciencedirect.com/science/article/B6V0C-475JVGT-2/2/2dc415be8050dc3f457218cad3ff9ee9>
- [12] Wiedermann, B. and Cook, W. R. (2006). Extracting Queries by Static Analysis of Transparent Persistence. .
- [13] Pohjalainen, P. and Taina, J. (2008). Self-configuring object-to-relational mapping queries. ACM International Conference Proceeding Series, 347, 7.
- [14] Philippi, S. (2005). Model driven generation and testing of object-relational mappings. Journal of Systems and Software, 77(2), 193--207. <http://dx.doi.org/10.1016/j.jss.2004.07.252>
- [15] Bruce, M. E. (2005). Uncovering Database Access Optimizations in the Middle Tier with TORPEDO. .

- [16] Carey, M. J., DeWitt, D. J., and Naughton, J. F. (1993). The OO7 Benchmark. In SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data, New York, NY, USA, 1993 (pp. 12--21). ACM.
- [17] Valdez, A. C. (2006). Entwicklung einer Benchmark und eines Werkzeugs zur Performance-Messung für Datenbankzugrisschichten. Masters thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen.
- [18] Alagöz, F. (2006). Konzeption und Implementierung eines Messverfahrens für den Performanz-Vergleich Konzeption und Implementierung eines Messverfahrens für den Performanz-Vergleich von Datenbankzugrisschichten. Masters thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen.
- [19] Smith, K. E. and Zdonik, S. B. (1987). Intermedia: A case study of the differences between relational and object-oriented database systems. SIGPLAN Not., 22(12), 452--465.
- [20] Cook, William, R., Greene, Robert, Linskey, Patrick, Meijer, Erik, Rugg, Ken, Russell, Craig, Walker, Bob, and Wittig, Christof (2006). Objects and databases: state of the union in 2006. In OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, New York, NY, USA, 2006 (pp. 926--928). ACM.
- [21] HibernateTeam (2009). Hibernate.org. Retrieved 18 dec 2009 from <https://www.hibernate.org/>.
- [22] Red Hat Middleware (2009). Hibernate reference documentation. Manning Publications Co., <http://docs.jboss.org/hibernate/stable/core/reference/en/html/>
- [23] hibernate forum (2009). SchemaExport not creating index for FK columns any more. Retrieved 17 dec 2009 from <https://forum.hibernate.org/viewtopic.php?t=924910>.
- [24] hibernate forum (21 jan 2004). hbm2ddl: Automatic creation of indexes on foreign keys. Retrieved 17 dec 2009 from <https://forum.hibernate.org/viewtopic.php?t=948998>.

# Master Thesis Software Engineering

Monday, January 11, 2010

## Appendixes





## Appendix A: Example mappings used in this research

In Figure 41 we indicated the mapping configuration we used in our research to create the table representations (indicated with the circle and number within). The recursive relationships are created in a similar way, but with both configurations in one object. For a description of these configurations see chapter 4.2 Mapping configuration or for more detail the Hibernate reference documentation [22].

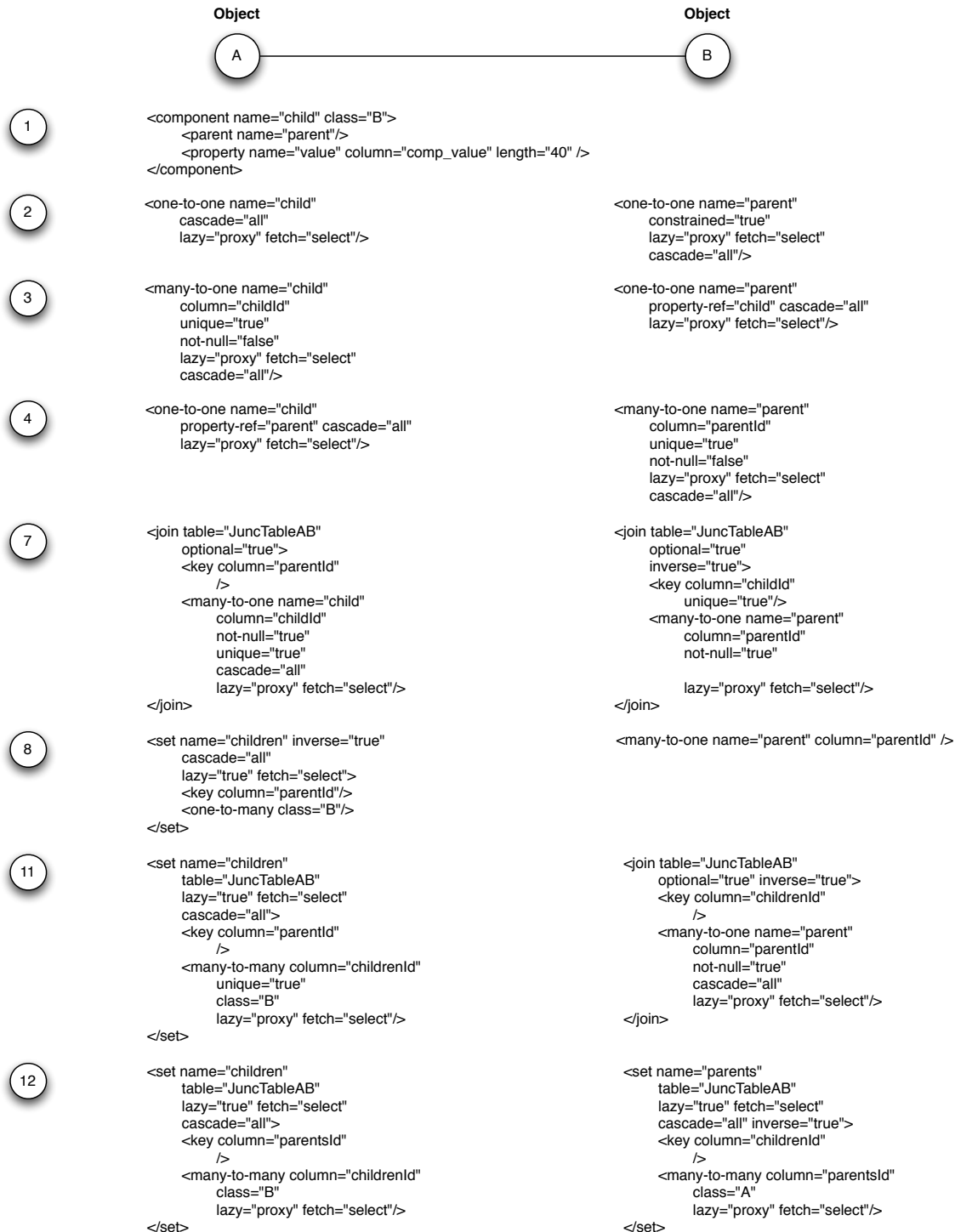


Figure 41: Example mappings used in this research

## Appendix B: Relation querying behaviour

In this appendix we worked out the relation querying of the type of relationships with different fetching strategies. For the legend of the diagrams used in this appendix, see Figure 42.

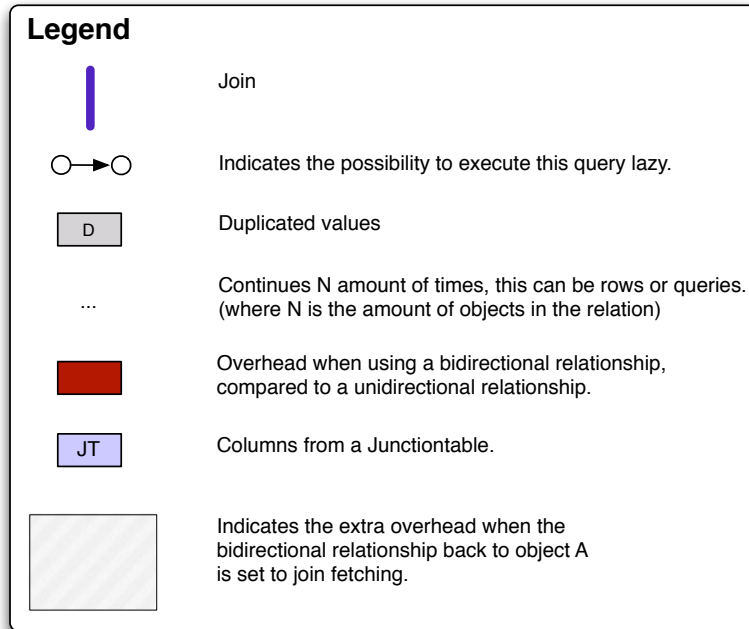


Figure 42: Legend behaviour diagrams

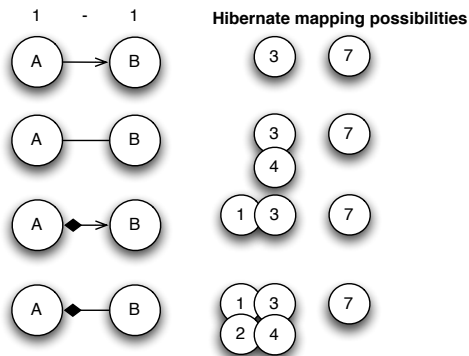
When looking at the performance we can distinguish three different factors that influence the performance: the amount of queries, type of query and size of the ResultSets. In the next part we compare these for each different table representations and fetching strategies. For convenience we also indicate what the behaviour is when the reference is set to lazy (if the object is not always needed, there can be a performance gain by not querying the object).

Underneath the TR (table representation) number we indicate the fetching strategy of the reference.

### One to one (non-recursive)

When comparing the non-recursive one-to-one relationships, there are five table representations and four object representations possible.

### Object reference representation



### Table representation

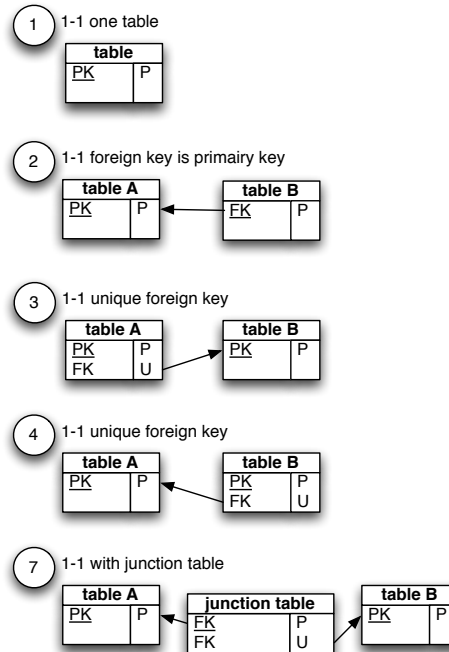


Figure 43: One to one (non-recursive), object and table representations

### Unidirectional aggregation

When a unidirectional aggregation relationship is used, there two table representations possible: one with and one without a junction table (see Figure 44).

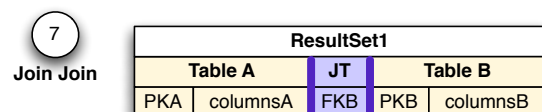
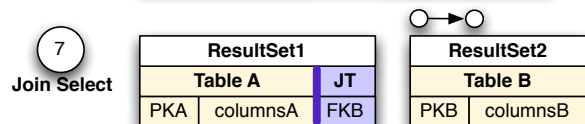
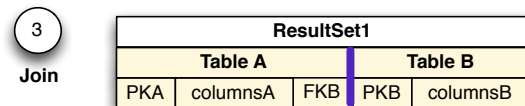
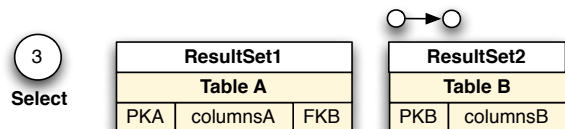
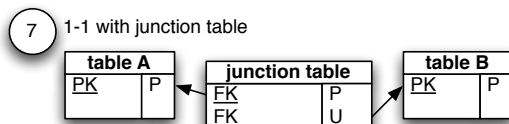
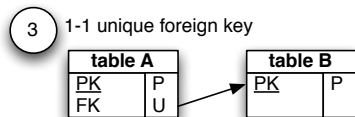
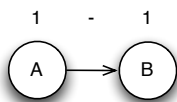
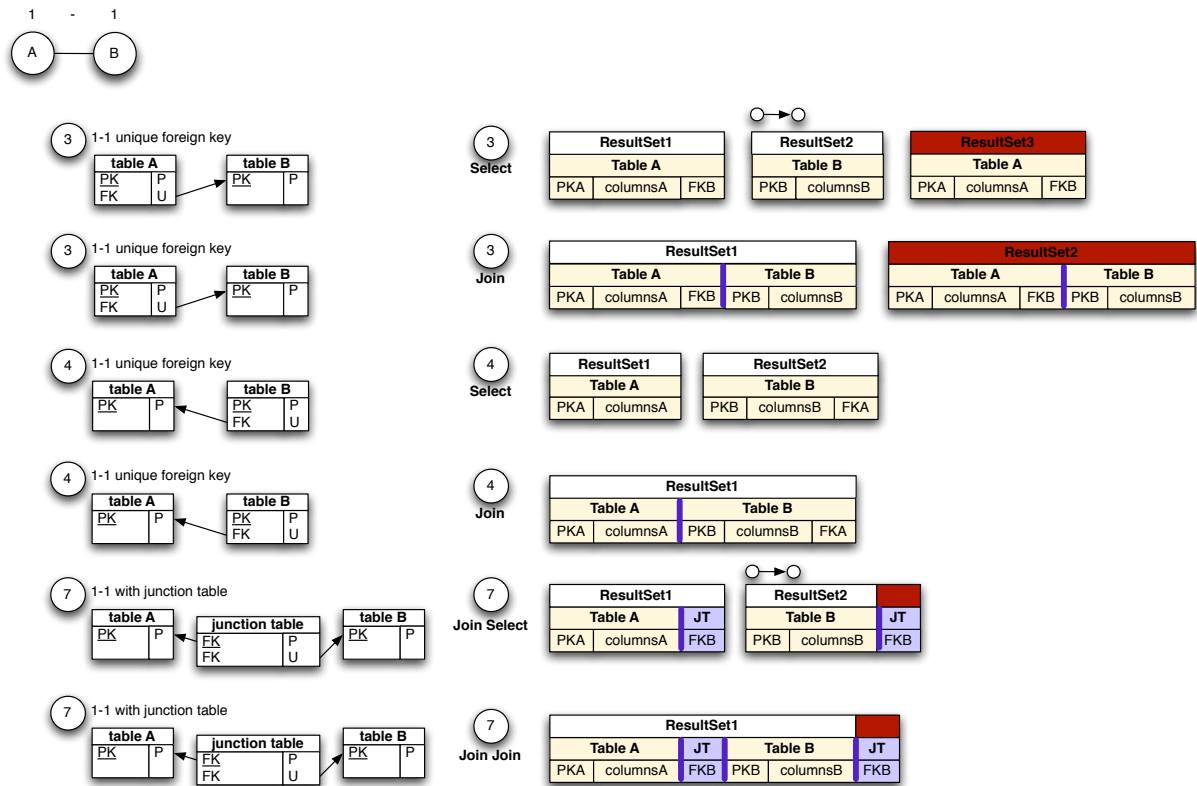


Figure 44: Unidirectional one to one aggregation

## Bidirectional aggregation

When a bidirectional aggregation relationship is used, there three table representations possible (see Figure 45).

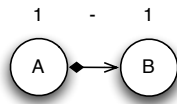


**Figure 45: Bidirectional one to one aggregation**

In the bidirectional relationships the fetching strategies are configured the same as the first object. Thus if object A has a join with the junction table and then a select with object B, object B will also have a join with the junction table and a select with object A.

## Unidirectional composition.

When an unidirectional composition relationship is used, there three table representations possible (see Figure 46).



1 1-1 one table

table	
<u>PK</u>	P

3 1-1 unique foreign key

table A		table B	
<u>PK</u>	P	<u>PK</u>	P
FK	U		

3 1-1 unique foreign key

table A		table B	
<u>PK</u>	P	<u>PK</u>	P
FK	U		

7 1-1 with junction table

table A		junction table		table B	
<u>PK</u>	P	FK	P	<u>PK</u>	P
		FK	U		

7 1-1 with junction table

table A		junction table		table B	
<u>PK</u>	P	FK	P	<u>PK</u>	P
		FK	U		

1

ResultSet1		
Table AB		
PKA	columnsA	columnsB

3  
Select

ResultSet1			ResultSet2	
Table A			Table B	
PKA	columnsA	FKB	PKB	columnsB

3  
Join

ResultSet1				
Table A			Table B	
PKA	columnsA	FKB	PKB	columnsB

7  
Join Select

ResultSet1			ResultSet2	
Table A			Table B	
PKA	columnsA	FKB	PKB	columnsB

7  
Join Join

ResultSet1				
Table A			Table B	
PKA	columnsA	FKB	PKB	columnsB

Figure 46: Unidirectional one to one composition

## Bidirectional composition

When an bidirectional composition relationship is used, there three table representations possible (see Figure 47).

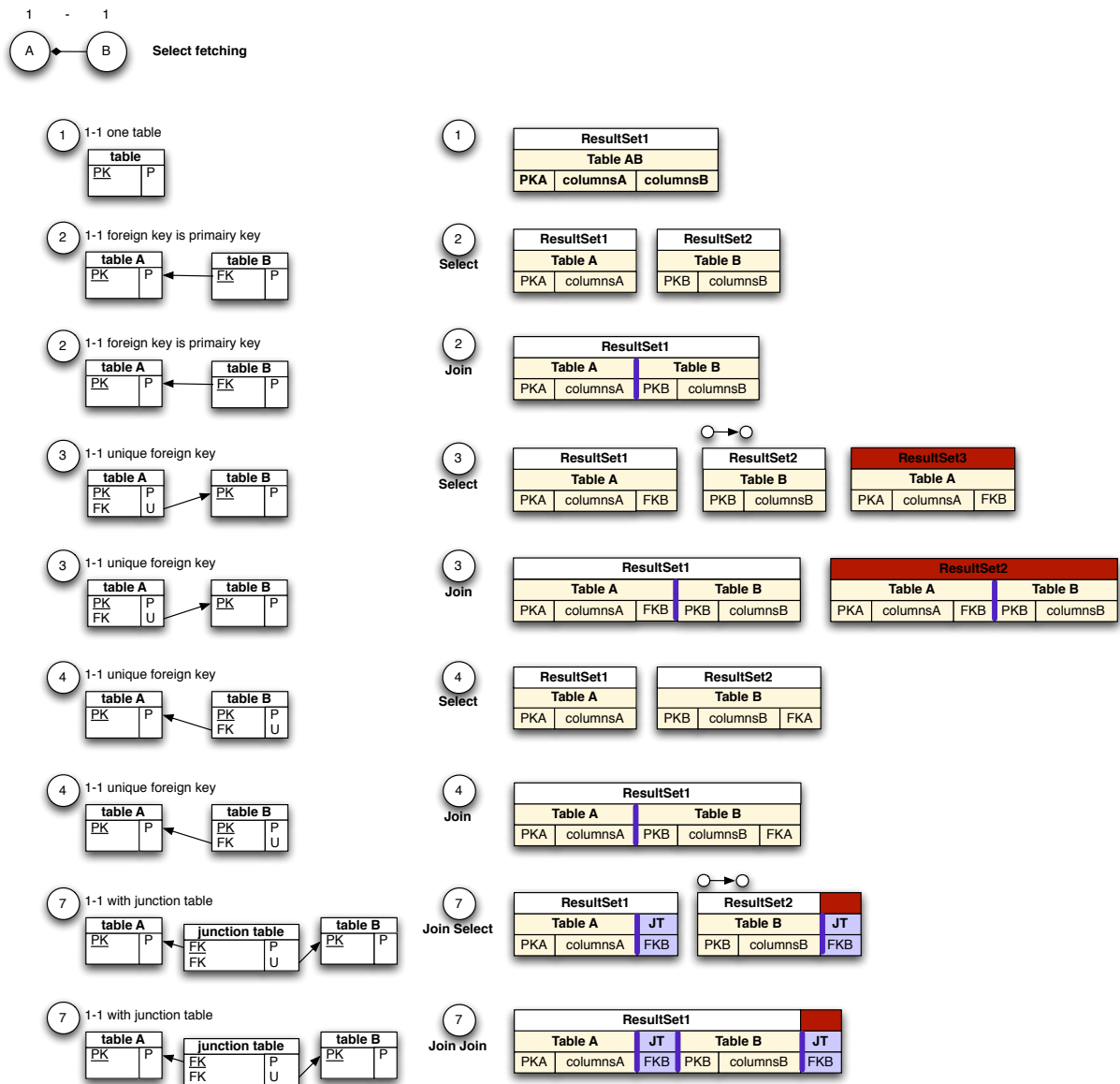
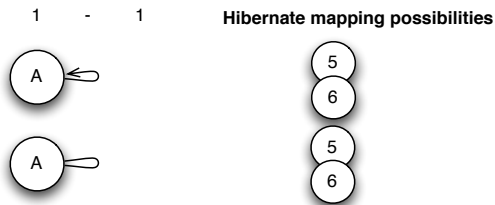


Figure 47: Bidirectional one to one composition

## One to one (recursive)

When comparing the recursive one-to-one relationships, there are two table representations and two object representations. See Figure 48.

### Object reference representation



### Table representation

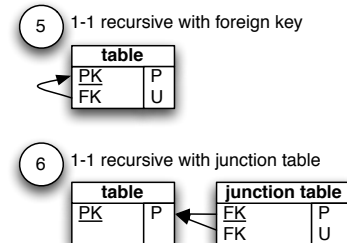
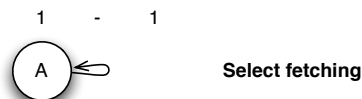


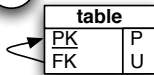
Figure 48: One to one (recursive), object and table representations

## Unidirectional aggregation

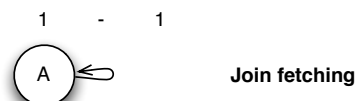
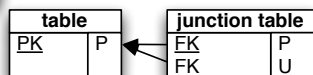
When a unidirectional relationship is used, there two table representations possible: one with and one without a junction table (see Figure 49).



5 1-1 recursive with foreign key



6 1-1 recursive with junction table



5 1-1 recursive with foreign key



6 1-1 recursive with junction table

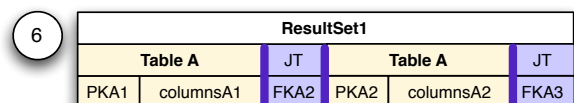
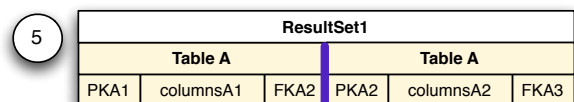
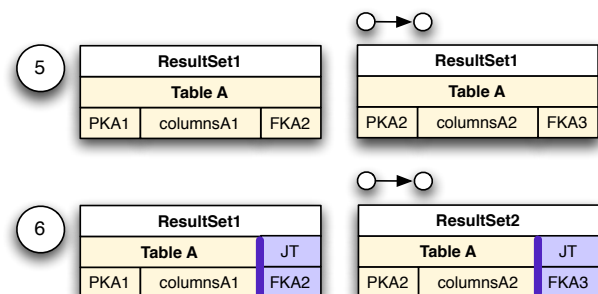
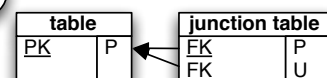


Figure 49: Unidirectional one to one recursive aggregation

## Bidirectional aggregation

With a recursive variant of the bidirectional aggregation there are no extra table representations possible compared to the unidirectional variant.

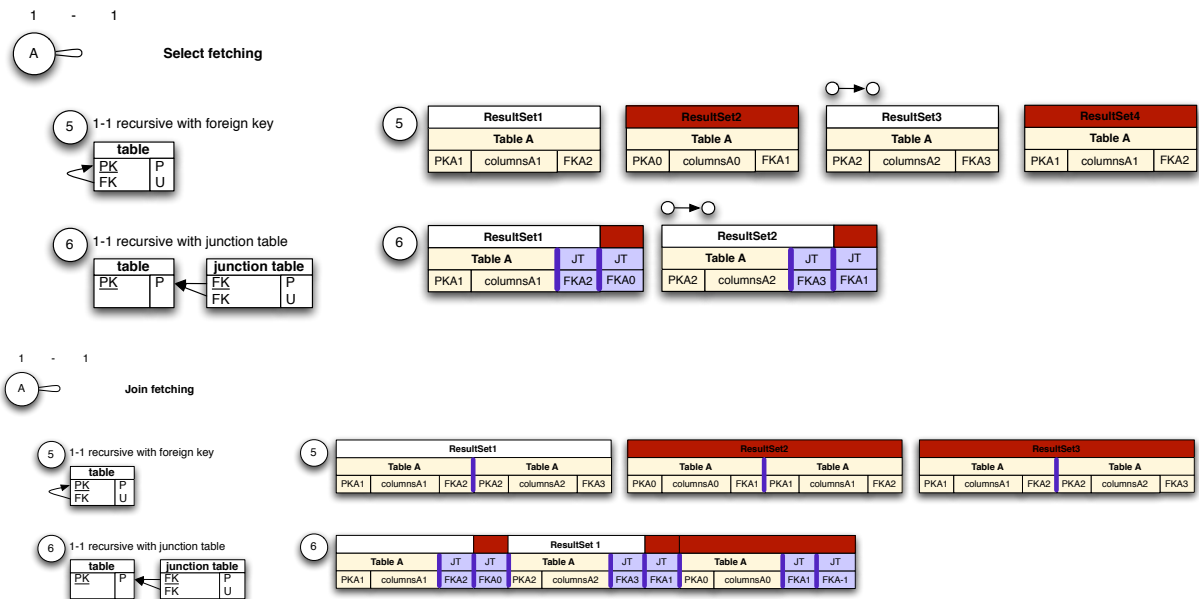


Figure 50: Bidirectional one to one recursive aggregation

## One to many (non-recursive)

When comparing the non-recursive one-to-many relationships, there two table representations and three object representations possible.

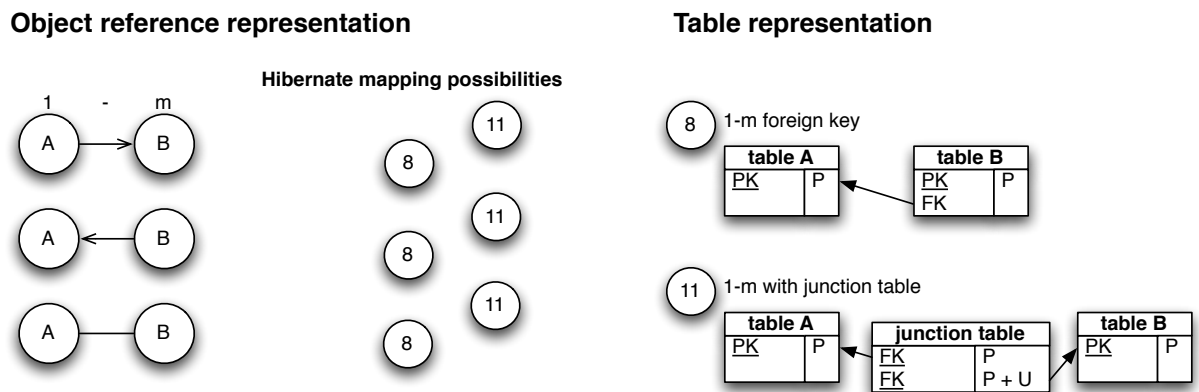


Figure 51: One to many (non-recursive), object and table representations

## Unidirectional aggregation

When a unidirectional aggregation relationship is used, there two table representations possible: one with and one without a junction table (see Figure 52). This relationship can be from either object A to B as well as from object B to A. The reference from B to A (many-to-one) is left out, because it can be compared to the one-to-one relationship (with table representations 4 and 7), but with the difference of a unique constrained or composition of the primary key. In the behaviour of Hibernate there will not be a difference, as the java object does not know if it belongs to a many or an one side.



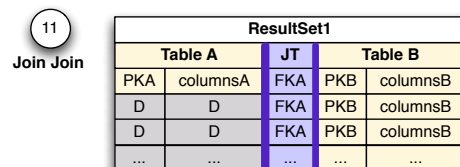
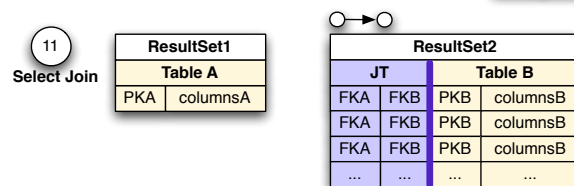
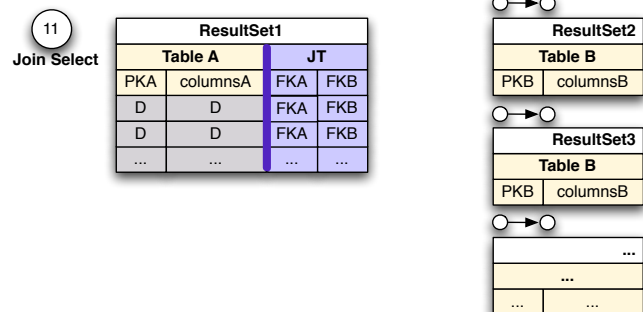
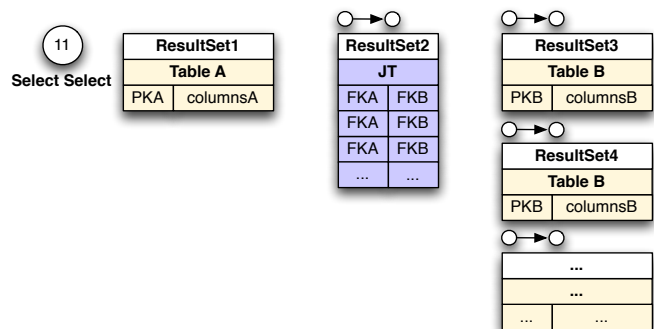
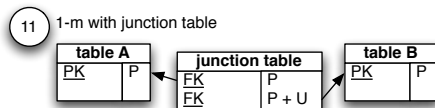
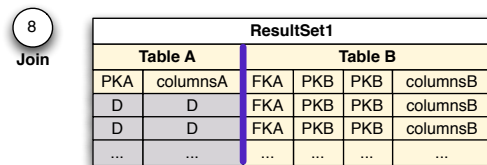
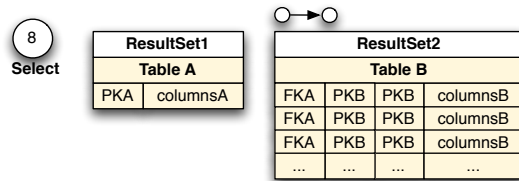
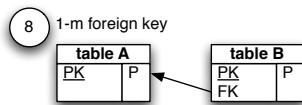
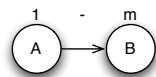



Figure 52: Unidirectional one to many aggregation

### **Bidirectional aggregation**

When a bidirectional aggregation relationship is used, there two table representations possible (see Figure 53).

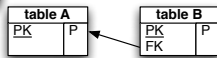
With this relationship the other side (the reference from B back to A) can be set a specific fetching strategy as well (besides the reference from A to B). The back reference of TR 8 is ignored, as this will never execute more queries joining or joining more tables together.

As the reference from B to A can be seen as an one-to-one relationship, the table B will always be joined with the junction table. The relationship between the junction table and table A will therefore be the only “back” reference that can be set to a specific fetching strategy. When this is set to select, object A can be retrieved from the cache (because we retrieved it before and we know the primary key). If this reference is set to join, the table will be joined with the previous query even if this object

is already known (in Figure 53 we indicated this extra joining with a  mark).



8 1-m foreign key



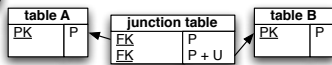
8 Select

ResultSet1		ResultSet2			
Table A		Table B			
PKA	columnsA	FKA	PKB	PKB	columnsB
FKA	PKB	PKB	columnsB	FKA	PKA
FKA	PKB	PKB	columnsB	FKA	PKA
FKA	PKB	PKB	columnsB	FKA	PKA
...	...	...	...	...	...

8 Join

ResultSet1					
Table A		Table B			
PKA	columnsA	FKA	PKB	PKB	columnsB
D	D	FKA	PKB	PKB	columnsB
D	D	FKA	PKB	PKB	columnsB
...	...	...	...	...	...

11 1-m with junction table



11 Select Select

ResultSet1	
Table A	
PKA	columnsA



ResultSet2	
JT	
FKA	FKB
FKA	FKB
FKA	FKB
...	...



ResultSet3			Table A	
Table B			JT	
PKB	columnsB	FKA	PKA	columnsA
PKB	columnsB	FKA	PKA	columnsA
PKB	columnsB	FKA	PKA	columnsA
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...

11 Join Select

ResultSet1			
Table A		JT	
PKA	columnsA	FKA	FKB
D	D	FKA	FKB
D	D	FKA	FKB
...	...	...	...



ResultSet2			Table A	
Table B			JT	
PKB	columnsB	FKA	PKA	columnsA
PKB	columnsB	FKA	PKA	columnsA
PKB	columnsB	FKA	PKA	columnsA
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...

11 Select Join

ResultSet1	
Table A	
PKA	columnsA



ResultSet2					
JT		Table B		JT	
FKA	FKB	PKB	columnsB	FKA	PKA
FKA	FKB	PKB	columnsB	FKA	PKA
FKA	FKB	PKB	columnsB	FKA	PKA
...	...	...	...	...	...

11 Join Join

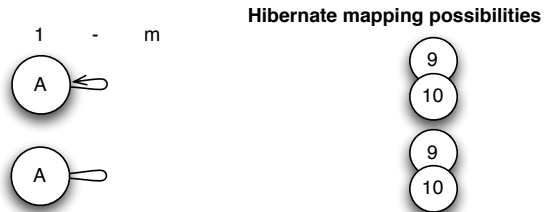
ResultSet1					
Table A		JT		Table B	
PKA	columnsA	FKA	PKB	columnsB	FKA
D	D	FKA	PKB	columnsB	FKA
D	D	FKA	PKB	columnsB	FKA
...	...	...	...	...	...

Figure 53: Bidirectional one to many aggregation

## One to many (recursive)

When comparing the recursive one-to-many relationships, there are two table representations and two object representations.

### Object reference representation



### Table representation

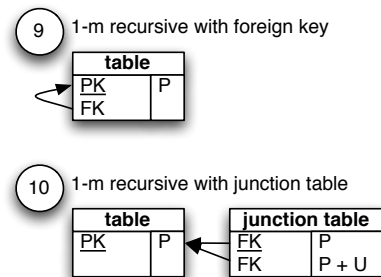
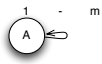


Figure 54: One to many (recursive), object and table representations

### Unidirectional aggregation

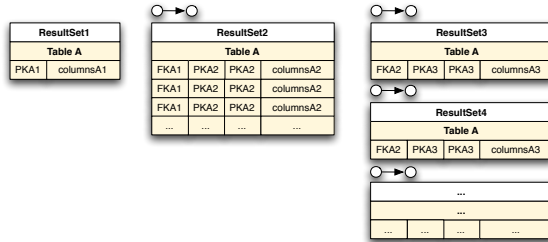
When a unidirectional aggregation relationship is used, there two table representations possible: one with and one without a junction table (see Figure 54).



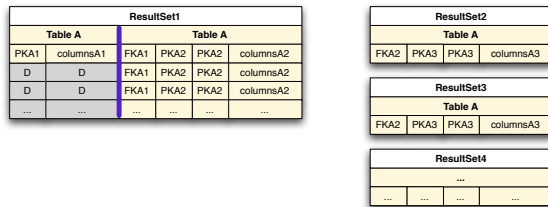
9 1-m recursive with foreign key



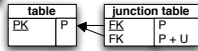
9  
Select



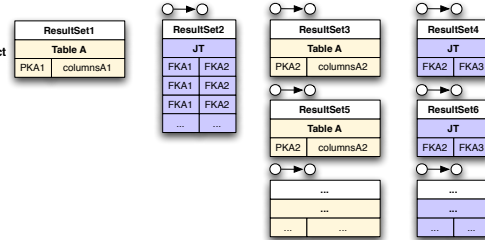
9  
Join



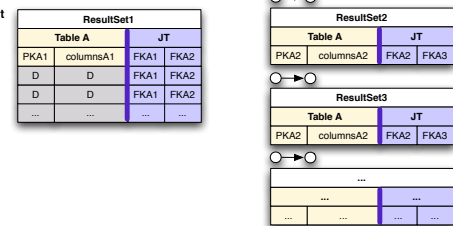
10 1-m recursive with junction table



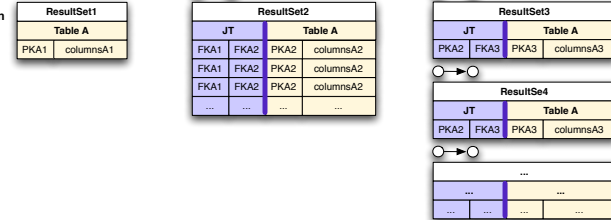
10  
Select Select



Join Select



Select Join



Join Join

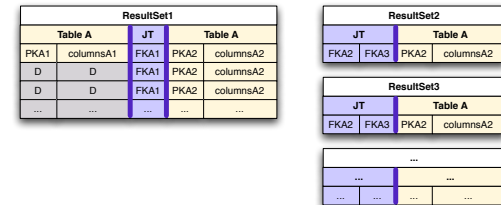


Figure 55: Unidirectional one to many aggregation

## Bidirectional aggregation

When a bidirectional aggregation relationship is used, there three table representations possible (see Figure 56).

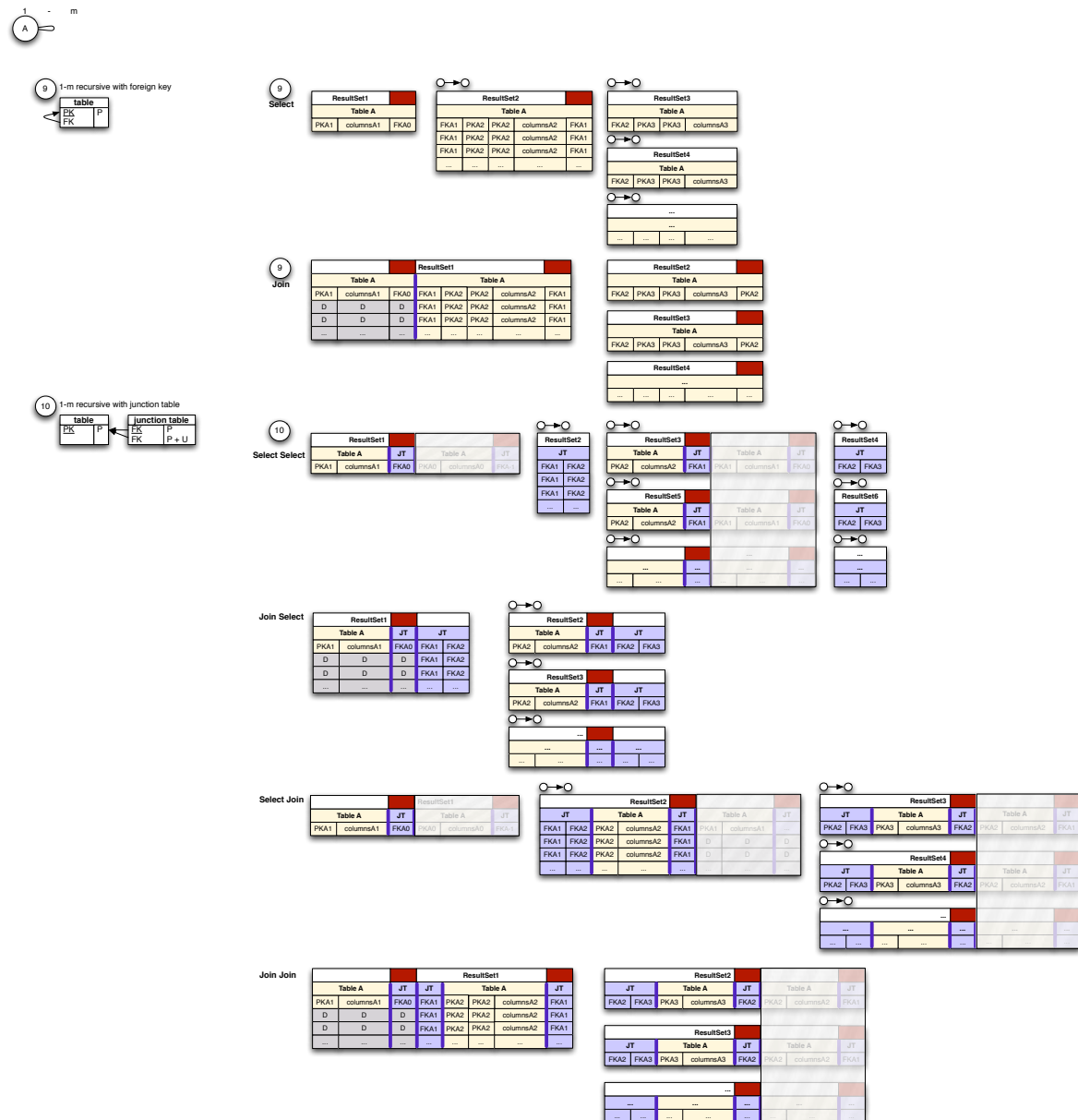


Figure 56: Bidirectional one to many aggregation

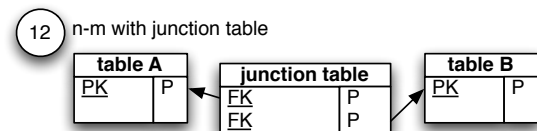
## Many to many (non-recursive)

When comparing the non-recursive many-to-many relationships, one table representation and two object representations are possible.

### Object reference representation



### Table representation



The unidirectional many-to-many relationship can be compared to the unidirectional one-to-many relationship (when choosing to use a junction table), except that the many side can now also be owned by several other objects. When looking at the table scheme this will only be a unique constraint or composition of the primary key. As the (right) many side does not have a reference back to the (left) many side, there will be no difference in behaviour of Hibernate when comparing with a one-to-many relationship.

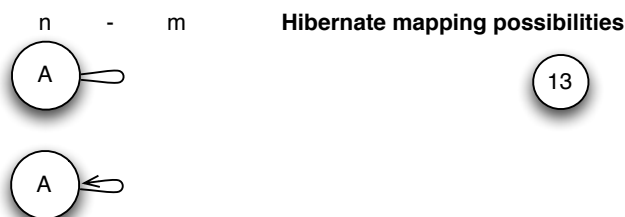
## Bidirectional aggregation

As the amount of queries is highly dependent on the amount of objects stored in the database and will increase very rapidly we will not create an overview of the relation querying of this relationship. The behaviour of this relationship can be created by concatenating multiple one to many relationships.

## Many to many (recursive)

When comparing the recursive many-to-many relationships, there are two table representations and two object representations. See Figure 57.

### Object reference representation



### Table representation

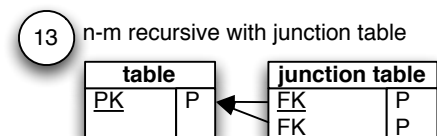


Figure 57: One to many (recursive), object and table representations

The recursive unidirectional many-to-many can also be compared with the recursive unidirectional one-to-many relationship.

## Bidirectional aggregation

As the amount of queries is highly dependent on the amount of objects stored in the database and will increase very rapidly we will not create an overview of the objects retrieved. The behaviour can be created by concatenating multiple recursive one to many relationships.