Masaryk University
Faculty of Informatics

Plausibility check of database operations speedup in system Perun using ORM tools

Bachelor's Thesis

Peter Jančuš

Brno, Fall 2018

MASARYKOVA UNIVERZITA
Fakulta informatiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Peter Jančuš

Program: Aplikovaná informatika

Obor: Aplikovaná informatika

Specializace: Bez specializace

Garant oboru: prof. RNDr. Jiří Barnat, Ph.D. (BcAP)

Vedoucí práce: RNDr. Martin Kuba, Ph.D.

Katedra: Katedra počítačových systémů a komunikací

Název práce: Ověření možnosti zrychlení databázových operací systému Perun pomocí ORM nástrojů

Název práce anglicky: Plausibility check of database operations speedup in system Perun using ORM tools

Zadání: Systém Perun slouží ke správě přístupu uživatelů k výpočetním prostředkům. Je implementován v jazyce Java a má vícevrstvou architekturu. Jeho vrstva persistence dat je v současnosti implementována pomocí SQL dotazů jejichž výsledky nejsou ukládány v paměti typu cache. Cílem práce je experimentálně ověřit, zda by nahrazení alespoň některých částí této vrstvy za ORM (Object-Relational Mapping) nástroje mohlo zrychlit některé obvyklé případy užití systému Perun. Předpoklad je, že některá data jsou čtena opakovaně a jejich uložení v paměti typu cache by omezilo počet přístupů k databázi. Vhodným ORM nástrojem by měla být kombinace Spring JPA s Hibernate.

Seznamte se s nástroji Spring JPA (Java Persistence API) a Hibernate. Zjistěte, které databázové dotazy jsou v systému Perun vykonávány nejčastěji opakovaně, a nahraďte je implementací pomocí Hibernate. Změřte rozdíl rychlosti provedení některých operací (například přidání skupiny s mnoha uživateli na výpočetní prostředek) v obou implementacích.

Literatura:

- Hibernate ORM documentation http://hibernate.org/orm/documentation/
- Spring Data JPA - Reference Documentation https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Zadání bylo schváleno prostřednictvím IS MU.

Prohlášení autora školního díla

Jméno, příjmení a UČO studenta:_____

Beru na vědomí, že *Masarykova univerzita* může na základě zákona (§ 35 odst. 3 a 4 autorského zákona č. 121/2000 Sb.) užít mou kvalifikační práci nebo jiné mé školní dílo, které jsem jako autor vytvořil ke splnění svých studijních povinností vůči této vysoké škole, a to k výuce nebo k její vlastní vnitřní potřebě nikoli za účelem přímého nebo nepřímého obchodního nebo jiného hospodářského prospěchu.

Vlastní vnitřní potřebou *Masarykovy univerzity* se rozumí užití díla nejen v původní podobě, ale též ve zpracované nebo jinak změněné podobě zahrnující též takové užití mého díla touto vysokou školou, které spočívá v zadání mého školního díla k dalšímu zpracování jinému studentovi této vysoké školy (členovi téže akademické obce) za účelem vytvoření další kvalifikační práce nebo jiného školního díla, které bude odvozené od mého díla při uvedení mého autorství, názvu mého díla a pramene; a to vše v souladu s rozvojem vzdělanosti na *Masarykově* univerzitě a zájmem této veřejné vysoké školy navazovat na výsledky mé práce a mé školní dílo dále rozpracovávat v téže akademické obci.

Okolnosti hodné zvláštního zřetele z mé strany, například zájem o vlastní další rozpracování své kvalifikační práce na *Masarykově univerzitě* nebo jinde, jsem povinen sdělit této vysoké škole prostřednictvím studijního oddělení nejpozději při odevzdání kvalifikační práce.

V Brně dne_____ _____

podpis studenta

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Peter Jančuš

Advisor: RNDr. Martin Kuba Ph.D.

Acknowledgements

I would like to thank my supervisor RNDr. Martin Kuba, Ph.D. and Perun team for all the support, time and advice they provided during my work on this thesis. I would also like to thank my family and friends for supporting me.

Abstract

This thesis is based on experimental testing of Hibernate - ORM tool with cache usage on database operations in the Perun system against the original JDBC queries. It describes this system and its data storage. Hibernate and its caching opportunities are described more in detail. It contains the test results and necessary specifications in case it will be integrated into the system.

Keywords

Perun, Hibernate, ORM, Second-level cache, Spring JPA

Contents

Introduction

The Perun system is suitable for the whole systems which need to manage users, their identities, associate users to groups or grant them access rights for several resources or services. This system keeps a lot of information about users, groups and many other entities in database, which is really extensive. In bigger instances it happens that data are read repeatedly therefore it is useless to access the database again and again even it could cause the overload of database.

Cache is a memory type with one of the fastest access to stored data and it allows to read this data repeatedly. Once the data are read from database, they are stored in cache as well and access to them is more quick than access to the database. This is the main premise on which this thesis stands on.

Relational database is collections of different tables which have predetermined form. The database user can access this data stored in tables. The Structured Query Language (SQL) is used for accessing these databases.

Object-oriented programming (OOP) is programming paradigm based on "objects". Each object can have many attributes usually with private access which means they can be accessed only from this object. Object's methods are used for access from any other part of code. This feature is called *encapsulation*. Another important features are *inheritance, polymorphism* or *abstraction*. OOP basically models real world principles. One of the OOP languages is Java, in which the Perun system is coded as well.

Object-Relational Mapping (ORM) is programming technique which is able to transform data from relational database to objects and back. One example is **Hibernate** which is main tool in this thesis. Hibernate allows to keep data which had been already read in *cache* memory.

This thesis is an experimental testing whether usage of Hibernate in the Perun system could speed up at least some database operations and help to reduce database load. First of all database operations have been analyzed to determine which are the most used in system followed by integrating this ORM tool with caching ability into system and testing execution time difference on the chosen operations.

In the first chapter I describe the Perun system in general, its main modules and the way how the data are stored. I describe the *JDBC, ORM* and Hibernate as well. The following chapter is split to initial state and requirements analysis. Third chapter is dedicated to *solution proposal* I describe Hibernate again, especially its main parts which are responsible for whole logic in code and caching possibilities and how they works. Analysis of *most frequent queries* and proposal of *Hibernate mapping* are situated at the end of the chapter.

The fourth chapter is whole about implementation, how I have integrated and configured Hibernate, its use in code and examples of queries provided by this tool. In following chapter I describe how I have tested and measured difference between JDBC and Hibernate, testing environment and finally evaluation of the tests. Last but one is basically discussion and I consider in it the possibilities how to integrate Hibernate to the Perun system in case the tests show as relevant. Final chapter is general conclusion of whole experimental testing.

1 Background

1.1 The Perun system

The Perun system is useful and complex tool able to manage users, groups and access to several services. Its development is ensured by CESNET [1] and Masaryk University [1] and users of this system have their own digital identities. According to [2] you can imagine digital identity as collection of information about digital subject. In our case digital subjects are the university employees or scientists using their institutional account. It is highly possible that a digital subject has more than one digital identity and not only does Perun system join these identities but it also can recognize the user as a specific digital subject in the case he or she logs in with any account.

A concept of VO (Virtual Organizations) for a services access management is used [3]. Every user has to ask for membership in appropriate VO which can have many groups. User is represented as member in groups. If user wants to have access to some services he needs to be a member of specific group which grants him access rights. Groups are allowed to have another subgroups, which can define different access rights.

There are many entities in Perun system. I have already mentioned a few of them, yet there are a way more of them. All of information about user like a name, title, email address and so on are kept in database. Each entity has a lot of specific information which is stored in database as well. There are listed entities stored in db:

- ActionType
- Attribute
- AuditMessage
- Ban
- Destination
- ExtSource

1. Czech Education and Scientific NETwork – `http://www.cesnet.cz/`

- Facility
- Group
- Host
- Member
- MembershipType
- Owner
- Resource
- ResourceTag
- Role
- Service
- User
- UserExtSource
- Vo

If we consider that Perun instances have on the order thousands of users and hundreds of groups it clearly shows the Perun databases are extensive.

1.2 Perun schema

The Perun system is divided into multiple modules where each one is dependant on at least one other module and has own specific function. In the Figure 1.1 you can see two main modules which I was working with in my thesis and connection between Perun Core module and database which is highly important for thesis.

"Perun RPC component is the main programmable interface of the Perun. Other components or external systems can communicate with Perun through the RPC" [4]. This module is split by methods managers, so as Perun Core, where each one is responsible for specific entity database operations.

Perun Core module implements main Perun logic and has multi-layered architecture divided into Facade, Service and DAO (Data Access Object) layer. It is common that multi-layered applications consist from many managers which handle CRUD [2] operations of appropriate entity. The most important for this thesis is DAO or sometimes called persistence layer which is responsible for entire communication and cooperation with some persist storage, usually and also in Perun it is relational database. Nowadays persistence layer is implemented in Perun by pure SQL queries. For this purpose the JDBC technology is used without any type of data caching.

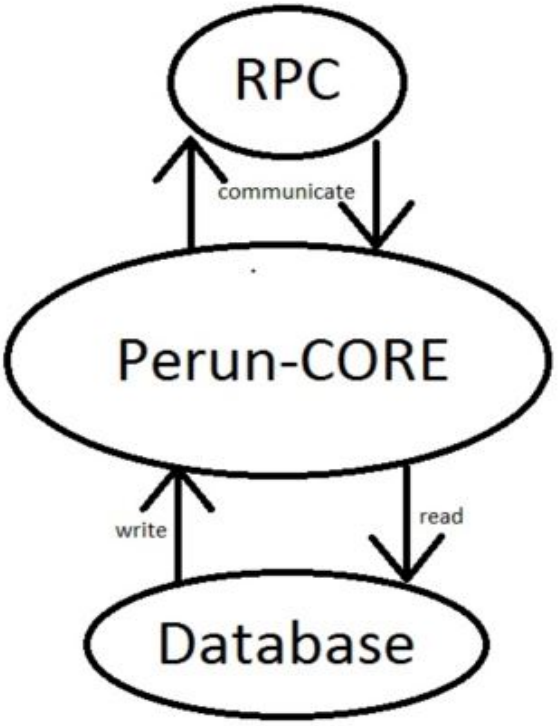

Figure 1.1: Schema

2. create, read, update and delete

1.3 Data storage in Perun system

The Perun system is able to persist data in two different relational database systems. First one is PostgreSQL[3] database which appears more often in Perun instances and the second one is Oracle[4] database [1]. As it will be described later in section 4.1, Hibernate has really simple and elegant solution for handling with various databases.

1.4 JDBC

JDBC is an abbreviation for Java Database Connectivity. This package offers basic interface for unified access to different types of database [5]. This purpose is achieved because of specific JDBC drivers. However it is still necessary to integrate SQL queries in specific database dialect which leads to application dependency on database. JDBC usually connects application with relational database, but it's not the only purpose, it is also dedicated for any data stored in "column" format.

1.5 Object-Relational Mapping

Object-Relational Mapping is programming technique which provides data transformation not only from relational database to object in object-oriented programming but also the other way from object to database format [6].

1.6 Hibernate

Hibernate is one of the most known ORM tool which works with Java objects. It is not necessary to know everything about SQL if you want to use Hibernate in your project for communication with database, but it is highly recommended to have at least basic knowledge about concepts which can lead to more simple usage of Hibernate [7]. Due to *Hibernate query language* (HQL) there is no dependency on specific database dialect which guarantees portability to another database

3. Postgre SQL - `http://www.postgresql.org/`
4. Oracle - `https://www.oracle.com/database/technologies/`

type without changing code or HQL. HQL works directly with objects not with tables, so there is needed to map entity classes represented by appropriate table in database. Hibernate offers two different ways how to map entities which I will describe later in chapter 3.3.

2 Analysis

2.1 Initial state

I have already indicated in 1.2 that Perun is using JDBC and pure SQL queries. JDBC is low-level API so that focuses only to construct queries and obtain a database communication.

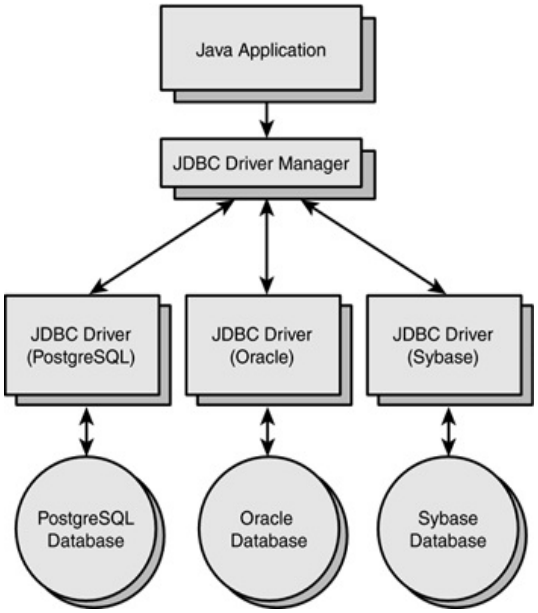

Figure 2.1: JDBC architecture: [8]

Usually there are four steps at interaction with database via JDBC. First one is to obtain a database connection. The JdbcTemplate[1] constructor with one attribute of data source, which carries information about database, is used in Perun for that reason. JdbcTemplate is basically superstructure from Spring JDBC library using design pattern, called template method, for easier work with JDBC. Following step is to create appropriate query and there is one issue that it is necessary to rename the column names in select clause, because there

1. JdbcTemplate - `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html`

are some columns with the same name in many tables. Of course it has an elegant solution, and that is the whole select clause is already prepared in variable. Whenever the new entity appears in Perun it is required to add all specifications for JDBC. Third step consists from the query execution and final step is to process the result which is specific for JDBC too. JDBC queries returns ResultSet [2] which needs to be processed manually by programmer. This is another disadvantage against Hibernate, but it is already managed in Perun for a long time so it doesn't play any role.

2.2 Requirements

This thesis is based on experimental testing between two different ways of database access. The main goal is whether an ORM tool such a Hibernate would have a positive impact on speed of some cases in Perun's persistence layer. The main premise is that some data are read repeatedly so their storage in a cache memory should reduce count of accesses to a database and therefore to speed up the whole process.

As I have described in chapter 1.3 Perun is able to persist data in two different databases so if testing will have positive impact it would be necessary to keep compatibility not only with PostreSQL but also with Oracle database as well. This is requirement which is really simple to handle from Hibernate site and it will be described in chapter 4.1.

The main task is to acquaint with Hibernate to figure out how it is working so I could successfully integrate Hibernate technology into Perun code, to properly configure it. It means that configuration should be placed in right part of code so the performance will be high as possible. It is necessary to properly configure Hibernate second-level cache as well. I need to find out whether Hibernate will be faster than JDBC in Perun system. This leads to sub task of the thesis to determine the most used queries in Perun, because it is best way how to reach a goal, to test it on these queries, which would have the greatest effect in general on database load.

---

2. ResultSet - `https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html`

3 Solution proposal

In this chapter I focus on more detailed Hibernate parts definition and its two caching opportunities. I also describe how I have decided to solve some issues like a mapping objects for Hibernate or how I analyzed queries for needed tests.

3.1 Hibernate architecture

Hibernate acts as a JPA provider and it implements Java Peristence API specifications. There are three basic interfaces which are needed for complete and reliable behaviour of Hibernate [9].

SessionFactory (*org.Hibernate.SessionFactory*)
The main, thread-safe and immutable, supplier for next important object in application - *Session*. Its equivalent in JPA is *EntityManagerFactory*. Creating of the *SessionFactory* is very expensive, so that the application should have only one associated *SessionFactory*. This interface is responsible for maintaining of many services that Hibernate uses across all *Session(s)* such as connections pools, transaction system integration or very important for us second level caches, etc [10].

Session (*org.Hibernate.Session*)
The main, single-thread and short-lived, run time interface which defines "Unit of work" according to [11]. Its equivalent in JPA is *EntityManager*. The main function is to provide basic *CRUD* operations for instances of mapped entity classes. *Session* wraps a JDBC *java.sql.Connection* and acts as a factory for *Transaction* instances [12].

Transaction (*org.Hibernate.Transaction*)
A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. *EntityTransaction* is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA) [13].

3.1.1 First level cache

Hibernate provides this caching by default and it isn't even possible to turn it off. First-level caching is associated with *Session* so that means while the Session is opened every data which has been read until it is closed, are stored in this cache. This data is accessible only from this specific Session and not from any other Session object nor from any other part of application [14]. This type of caching is not exactly what is needed to reach a necessary goal - to keep data in cache after closing the *Session*.

3.1.2 Second level cache

This caching type is associated with *SessionFactory* and it means all data stored in this cache is shared by all sessions which were created by the same session factory. This data are alive in second-level cache while SessionFactory is alive too. Due to SessionFactory has usually just one instance in application, the data will be kept in this cache which I consider very useful for this thesis. When an entity instance is looked up by its id, it is returned from first-level cache (in case that an instance is already present in this cache). If it is not found there then it is returned from second-level cache (in same case as first-level cache). Otherwise, the entity is loaded from database and new instantiated entity is returned [14].

3.2 Most frequent queries analysis

Due to a large number of queries in Perun, it was necessary to analyse them, so I could find out which are the most often called. The simplest way to detect them was to turn on logging of SQL queries and then let the python script to count it. Perun system is used in many instances so I have chosen the most loaded one, so I could easily determine queries which I will compare. A total amount of logged queries was 3140 and I have decided to choose queries which count of calls was above one hundred thousand and those which are not primary working with attributes[1].

1. Attribute - The entity in Perun `https://perun-aai.org/documentation/technical-documentation/core/index.html`

Twenty-nine queries met the count requirements and only fifteen of them was working with different entities than attributes. The first problem occurs here, because I have found out that there are tables in Perun database which have no corresponding entity class in Perun-Core module. I have read many articles and lines of documentation to determine if it is possible that Hibernate could handle it. Unfortunately there is no possibility to work with this kind of tables, because Hibernate is entity related tool. I explain later in 6.1 what is needed for Perun implementation to use Hibernate for every database operation.

3.3 Mapping

According to ORM technology and its principle, it was needed to map various entities of Perun system. Hibernate offers two different solutions how it could be done. First one is using annotations to provide mapping information. It is based on adding specific annotations (which have some meaning) for each entity class. If I chose this one, I would have to change the code in many classes and it is something that I wanted to avoid. The second solution is that "Hibernate uses the mapping metadata to determine how to load and store objects of the persistent class. The Hibernate mapping file is one choice for providing Hibernate with this metadata" [15]. The form of this metadata is stored in xml file. Due to how extensive the Perun system already is, I consider using xml mapping more simple.

There are few cases that entity has different attribute type than corresponding database table has (e.g. status / membership / ...) and I have decided to handle this with **AttributeConverter<X,Y>** interface from *javax.persistence*. "A class that implements this interface can be used to convert entity attribute state into database column representation and back again" [16].

4 Implementation part

This chapter is the most extensive because I show the way how I have integrated Hibernate to the Perun system with code samples, Hibernate configuration or example of mapping file. Also there will be specific implementation of database operations using HQL.

4.1 Hibernate configuration

By the time of compiling the application Hibernate requires to know much information to be successfully built. First of all the connection settings, related to a database, are really important. If Hibernate should fulfill its main functionality, which is work with a database, it needs to have the connection with one. If there is some functionality which is above the basic one this is a place where to put specifications about turning it on or whatever is needed. This kind of information for Hibernate is specified in a single property which usually has one attribute name and value inside this property which is basically the specification of requested functionality.

Hibernate also needs to know where to find information about mapping objects which is easily done by *property* name *mapping* with attribute resource or class. It depends on what kind of mapping, which I have mentioned earlier in 3.3 Mapping. In case an annotations are used, the class is specified in mapping. In my implementation I used xml mapping, therefore I specify the paths to files one by one in resource attributes.

I used for storing all such information the xml file with specific name, which was necessary, "Hibernate.cfg.xml". Many properties take their default values. This file is kept in the root directory of your application's classpath [14].

Each such a configuration is inside a tag "session-factory" and it means, that every time the new *SessionFactory* is created it has to be done with all these specifications consequently the new database connection is provided too, as a result is very expensive creation of *SessionFactory* which makes it a long-time living object. Therefore it is usually created once in application.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE Hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://Hibernate.sourceforge.net/Hibernate-
    configuration-3.0.dtd" >
<Hibernate-configuration>
    <session-factory>
        <!-- Database connection settings-->
        <property name="connection.driver_class">
        org.postgresql.Driver</property>
        <property name="connection.url">
        jdbc:postgresql://localhost:5432/perun</property>
        <property name="connection.username">username</property>
        <property name="connection.password">password</property>
        <!-- JDBC connection pool (use the built-in)-->
        <property name="connection.pool_size">1</property>
        <!-- SQL dialect-->
        <property name="Hibernate.dialect">
        org.Hibernate.dialect.PostgreSQL95Dialect</property>
        <!-- Echo all executed SQL to stdout-->
        <property name="show_sql">true</property>
        <property name="Hibernate.cache.region.factory_class">
        org.Hibernate.cache.jcache.JCacheRegionFactory</property>
        <property name="Hibernate.javax.cache.provider">
        org.ehcache.jsr107.EhcacheCachingProvider</property>
        <property
            name="cache.use_second_level_cache">true</property>
        <property
            name="Hibernate.cache.use_query_cache">true</property>
        <property name="net.sf.ehcache.configurationResourceName">
        ehcache.xml</property>
        <property name="hbm2ddl.auto">validate</property>
        <!-- Mapping resources -->
        <mapping resource="HibernateMapping/Facility.hbm.xml" />
        <mapping resource="HibernateMapping/User.hbm.xml" />
        <mapping resource="HibernateMapping/ExtSource.hbm.xml" />
        <mapping
            resource="HibernateMapping/UserExtSource.hbm.xml" />
        <mapping resource="HibernateMapping/Service.hbm.xml" />
        <mapping resource="HibernateMapping/Vo.hbm.xml" />
        <mapping resource="HibernateMapping/Group.hbm.xml" />
    </session-factory>
</Hibernate-configuration>
```

Code sample 4.1: Hibernate configuration

connection.driver_class specifies the JDBC driver which tells Hibernate type of database to connect it.

connection.url is the jdbc url to specific database instance.

connection.username and *connection.password* specify the username and password. For testing purpose it was much simpler to keep it there. In case Hibernate shows as a good choice and it will be used in Perun system it will be needed to hide at least the password, because the Perun is open-source project and it is not safe to show password on GitHub. It could be easily stored in properties file on virtual machine where the system is running which could be read in code and then added as property while configuring.

connection.pool_size limits the maximum number of connections to database.

Hibernate.dialect is responsible for transformation HQL to appropriate SQL for chosen database. In collaborate with *connection.driver_class* they are responsible for flexibility with various databases.

show_sql as it is in note it writes all SQL queries to stdout. It is set on true because it was needed to recognize how many times was Hibernate accessing the database while I was testing. Simply to check if the second-level cache works well.

hbm2dl.auto this property automatically validates or exports schema DDL to the database when the SessionFactory is created [13]. It is set to *validate*, because I was testing only read methods. In case the write into database is needed, this property must be set to *update*.

The other properties are used for enabling second-level cache which meaning I'll explain later in specific section 4.4.

The paths to *xml mapping files* are situated at the end of configuration file.

4.2 Mapping objects

I have decided to create a single file for each entity, which I wanted to re-implement by Hibernate. All of them are very similar, but there were some specific cases which are described below.

```
<?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>
    <class name="cz.metacentrum.perun.core.api.Facility"
        table="facilities" >

        <cache usage="read-only"
            region="cz.metacentrum.perun.core.api.Facility" />
        <id name="id" type="org.Hibernate.type.IntegerType">
            <column name="id" not-null="true"/>
        </id>
        <property name="name"
            type="org.Hibernate.type.StringType">
            <column name="name" length="128" not-null="true" />
        </property>
        <property name="description"
            type="org.Hibernate.type.StringType">
            <column name="dsc" length="1024" />
        </property>
        <property name="createdAt" >
            <column name="created_at" not-null="true" />
            <type name="converted::cz.metacentrum.perun.core.api.
                AttributeConverters.TimeStampConverter" />
        </property>
        <property name="createdBy"
            type="org.Hibernate.type.StringType">
            <column name="created_by" length="1300"
                default="user" not-null="true" />
        </property>
        <property name="modifiedAt" >
            <column name="modified_at" not-null="true" />
            <type name="converted::cz.metacentrum.perun.core.api.
                AttributeConverters.TimeStampConverter" />
        </property>
        <property name="modifiedBy"
            type="org.Hibernate.type.StringType">
            <column name="modified_by" length="1300"
                default="user" not-null="true" />
        </property>
        <property name="createdByUid"
            type="org.Hibernate.type.IntegerType">
```

```
            <column name="created_by_uid" />
        </property>
        <property name="modifiedByUid"
            type="org.Hibernate.type.IntegerType">
            <column name="modified_by_uid" />
        </property>

    </class>
</Hibernate-mapping>
```

Code sample 4.2: Mapping by xml file

All XML mappings should declare the doctype shown. The actual DTD can be found at the URL above, in the directory Hibernate-x.x.x/src/org/Hibernate , or in Hibernate3.jar. Hibernate will always look for the DTD in its classpath first **[hibMapBK]**

Inside Hibernate mapping I have defined property class. The name attribute names the entity class, which I want to map. In case it situates in different module it is necessary to specify the whole path to *Entity*. The *table* attribute names the appropriate database table which contains data for this entity. Instances of the *Facility* class are now mapped to rows in the *facilities* database table.

Element name *cache* indicates that caching of this mapped *entity* is allowed. *Usage* specifies the caching strategy: *transactional, read-write, nonstrict-read-write* or *read-only*. I have chosen read-only strategy because the application needs just to read most of mapped entities in my implementation. In case the update is necessary the other three options are available. The main difference between them is on transaction level [17].

Hibernate uses the property named by the *id* element to uniquely identify rows in the table. The *id* element here names the *facility_id* column as the primary key of the *facilities* table. It also identifies the id property of the *Facility* class as the property containing the identifier value [18].

```
<many-to-one name="extSource" unique="true"
    class="cz.metacentrum.perun.core.api.ExtSource">
    <column name="ext_sources_id" not-null="true" />
</many-to-one>
```

Code sample 4.3: Cardinality mapping

There was also a case of unidirectional cardinality[1] in *UserExtSource* mapping file which has to be mapped as well. For this purpose the attribute *unique* must be set to true. In *class* attribute you specify entity class which the mapped entity must contain.

The rest of attributes are defined as a property with its name, which corresponds to entity's attribute name so as its type. Inside property is defined column and its name which represents column from the database table. Then follow next options which I have defined such a not-null or default which Hibernate is using while creating or updating tables. I haven't needed them for testing purposes, but I did it as an example.

As you can see properties *createdAt* and *modifiedAt* have the specific type. As I have mentioned earlier in 3.3 if entity and its representation in database has not compatible data types, conversion is needed. In type name must be written "converted::" followed by path to particular *AttributeConverter*.

4.3 Second-level cache configuration

First of all the dependency into *pom.xml* file needs to be added for cache provider according to chosen one. Then *region.factory_class* and *cache.provider* need to be specified in configuration file. The property *cache.use_second_level_cache* must be set to true and if we want to enable also queries caching the property *Hibernate.cache.use_query_cache* as well. These specifications are situated inside the *Hibernate.cfg.xml*.

I have decided to choose combination of JCache API with EhCache-Provider. The JCache API (a.k.a. JSR-107[2]) is a Java specification for

1. Cardinality in data modeling - `https://en.wikipedia.org/wiki/Cardinality_(data_modeling)`
2. JSR-107 - `https://jcp.org/en/jsr/detail?id=107`

temporary, in memory caching of Java objects, including object creation, shared access, spooling, invalidation, and consistency across JVM's. Ehcache[3] is open source and most widely used java based cache, which can be integrated with popular libraries and frameworks. Ehcache 3.x provides an implementation of JCache specification, which is fully compliant with the JCache API.

4.4 Implementation

Configuration of everything what is necessary has been already set so now I will show the implementation of my solution. I have already explained earlier in 1.2 two of Perun modules which I was working with. Tests will be described later, but I just need to mention, that I have decided to test the specific methods by HTTP GET REQUESTS which are handled by *Perun RPC* module. This is place where I implement creation of *SessionFactory* and to open *Session* and commits the *transaction*. Inside each specific manager are also the specific methods called from *Perun Core* module.

```
public class HibernateUtils {
  private static SessionFactory sessionFactory;
  public synchronized static SessionFactory getSessionFactory() {
     if (sessionFactory == null) {
        StandardServiceRegistry standardRegistry = new
            StandardServiceRegistryBuilder().configure().build();
        Metadata metadata = new
            MetadataSources(standardRegistry).getMetadataBuilder()
            .build();
        sessionFactory =
            metadata.getSessionFactoryBuilder().build();
     }
     return sessionFactory;
  }
}
```

Code sample 4.4: SessionFactory creation

I have created this class to ensure, that *SessionFactory* will be created exactly once, because of its high time of creation. Until StandardSer-

3. Ehcache - `http://www.ehcache.org/`

viceRegistry is built, there is allowed to read username and password from file and set as property, as I have mentioned earlier in 4.1.

```
getUserById {
  @Override
  public User call(ApiCaller ac, Deserializer parms) throws
      PerunException {

    long startTimeMs = System.currentTimeMillis();

    Session session =
        HibernateUtils.getSessionFactory().openSession();
    Transaction tx = session.beginTransaction();
    ac.getRpcSession().setSession(session);
    ac.getSession().setSession(session);

    User user = ac.getUserById(parms.readInt("id"));

    tx.commit();
    session.close();

    long elapsedMs = System.currentTimeMillis() - startTimeMs;
    Integer id = parms.readInt("id");

    try {
      FileWriter fileWriter = new
          FileWriter("/home/tests/test.txt", true);
      PrintWriter printWriter = new PrintWriter(fileWriter);
      printWriter.println("[getUserById " + id.toString() + "]
          time elapsed millis " + elapsedMs);
      printWriter.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }

    return user;
  }
},
```

Code sample 4.5: Get User by id in Perun RPC

This is an instance of call one of tested methods also with time measurement. The main Hibernate work starts by opening a new *Session* from *SessionFactory* which has been got from *HibernateUtils* class. *Session* begins a new *Transaction*, which allows the application to define "units of work", Session is set and stored in *PerunSession*[4] class which is accessible later from DAO layer in Perun Core module, what is required for creating the query. It is not necessary to show all the details, but *ApiCaller* calls this method from Perun Core module where the persistence logic is situated. After ApiCaller returns the User the transaction is commited and Session is closed. At the end of method is just provided write into file, which is done after I measured the time, so it has no effect on tests.

```
@Override
public User getUserById(PerunSession sess, int id) throws
    InternalErrorException, UserNotExistsException {
  try {
     Session session = sess.getSession();
     User u = session.find(User.class, id);
     return u;
  } catch (NoResultException ex) {
     User user = new User();
     user.setId(id);
     throw new UserNotExistsException(user);
  } catch (RuntimeException ex) {
     System.out.println(ex.getCause());
     throw new InternalErrorException(ex);
  }
}
```

Code sample 4.6: Session.find query example

```
@Override
public List<UserExtSource> getUserExtSources(PerunSession sess,
    User user) throws InternalErrorException {
  try {
     Session session = sess.getSession();
     Query<UserExtSource> query;
     if (sess.getUserExtSourceQuery() == null) {
        query = session.createQuery("select u From UserExtSource
            u where u.userId= :uId", UserExtSource.class)
```

4. PerunSession - Encapsulates context of user's interaction with the Perun system.

```
            .setCacheable(true);
         sess.setUserExtSourceQuery(query);
      } else {
         query = sess.getUserExtSourceQuery();
      }
      List<UserExtSource> userExtSources = query
         .setParameter("uId", user.getId())
         .getResultList();
      return userExtSources;
   } catch (RuntimeException e) {
      throw new InternalErrorException(e);
   }
}
```

Code sample 4.7: Session.createQuery example

These queries are created on DAO layer. In "find" sample there is nothing interesting, it simply finds the Entity via id, but inside method *getUserExtSources* I have created a query and store it in *PerunSession*. This ensures that a query is created only once, which is provided by if condition. Not only enabling query caching in *Hibernate.cfg.xml* is required to provide second-level caching, but also setCacheable on true while query is creating. For *Session.find* method this is already set by default.

5 Testing and valuation

This chapter is dedicated to the evaluation of Hibernate's impact on database operations. I have measured time difference using Hibernate between first query and the second one when an object is already saved in cache memory. The second test consists of two get methods which correspond to code samples 4.6 and 4.7. These methods compare times for getting one object 10 000 times in a row for 100 times between Hibernate and JDBC queries. All results are represented in graphs for clarity. As it has been already shown in code sample 4.5 the time is measured in Perun RPC module and it contains the whole time duration by all layers in Perun Core module. The conditions were same for Hibernate and also JDBC queries so it has no impact on the final result.

5.1 Testing environment

I have tested methods and I have got following results on my own laptop. There has been Ubuntu 17.10 installed and PostreSQL database, version 9.6.9 in standard configuration. Machine has CPU: Intel(R) Core(TM) i7-4720HQ @ 2.60Hz and 8GB RAM. Any useless processes were not running during the tests.

5.2 Difference with cache

This test has been composed of one single query to get User by its id before and after caching the User. First of all I have called the query with a specific user's id and then again with the same id. This test has tested the second-level cache related to *Session*, after first call the *Session* has been closed and in second call the new *Session* has been opened. Both times have been measured and compared in the graph. This test has been called fifty times each time on a different user. Before testing, I have called another query 50 000 times to avoid deflection related with Java starting process.

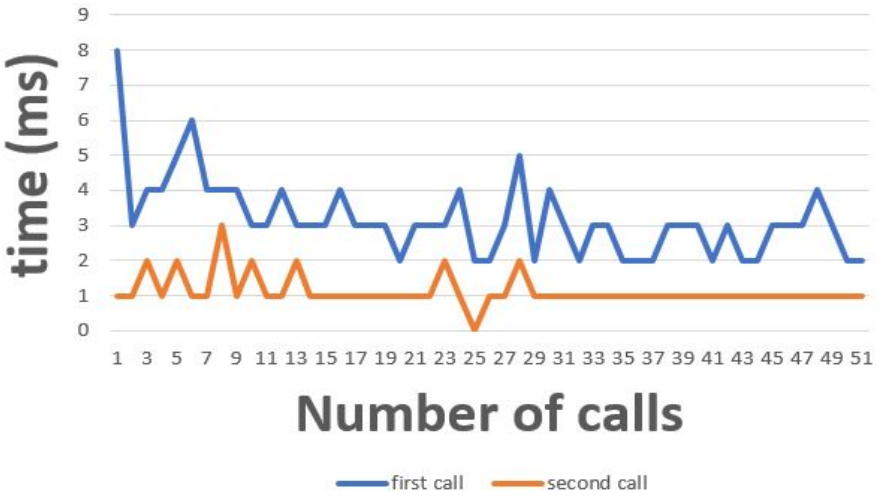

Figure 5.1: getUser - every call is one query

As the graph has shown the second-level cache has made the getting User method on average 2.92x faster and average time difference has been 2.06 ms. This test has shown that usage of Hibernate second-level cache makes the single query faster indeed.

5.3 Hibernate and JDBC queries comparison

In this section, there are shown two different compare tests between Hibernate (with second-level cache integrated) and JDBC. Both have been composed from 10 000 queries which have been called 100 times in a row. One test again to get user by its id and second one to get user's external sources. For the same reason as in the first test, 50 000 queries have been called before testing as well.

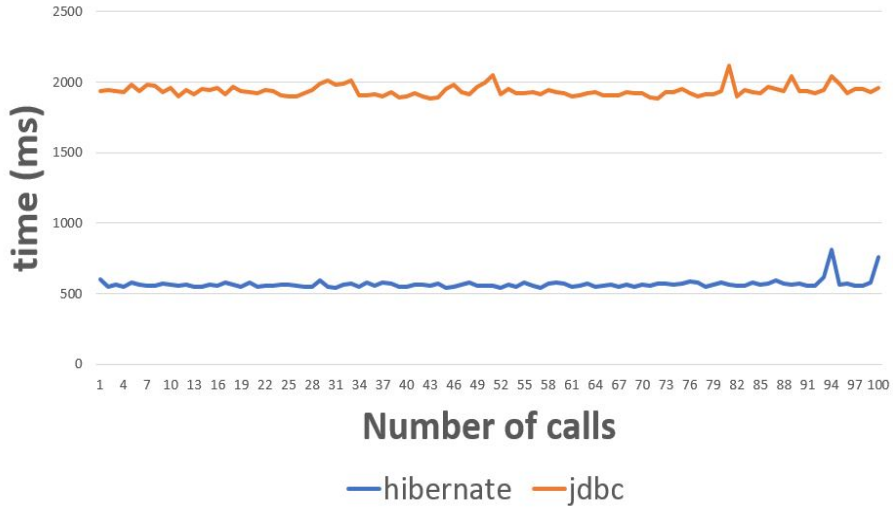

Figure 5.2: getUser - every call is 10 000 queries

In case getting User by id the Hibernate has been on average 3.42x faster over JDBC and average time difference has been 1370 ms.

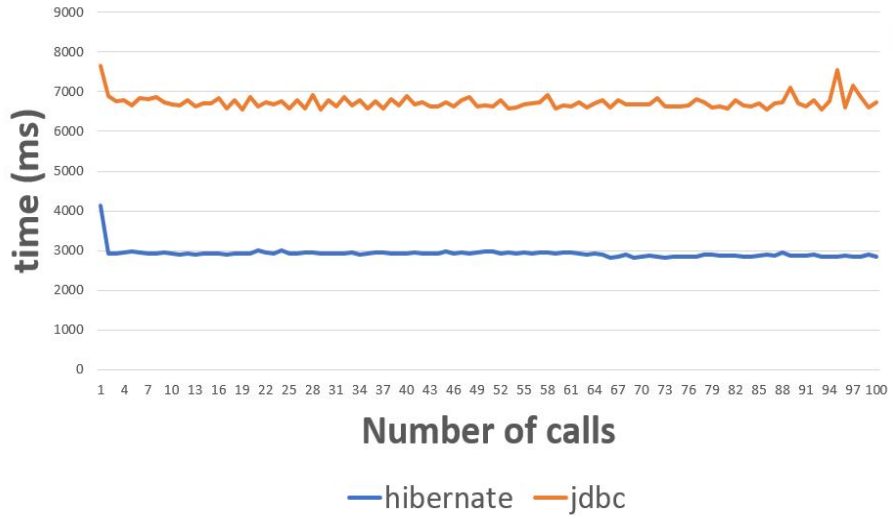

Figure 5.3: getUserExtSources - every call is 10 000 queries

In case getting User's external sources the Hibernate has been on average 2.3x faster over JDBC and average time difference has been 3809 ms.

Both tests have clearly shown that Hibernate usage at least for read database operations in Perun system has a significantly positive impact. If we consider that in time of logging SQL queries, which was almost one day, the get user by id method has been called almost 13 million times and get user's external sources over 1 million times then decrease of database load could be significant. Database operation's speed up is dependant on call type so it is not constant.

6 Discussion

The tests have shown that Hibernate with cache functionality is faster than pure JDBC and in case the Hibernate will be integrated into Perun system there are some issues and it is necessary to be aware of them. There are cases of database operations in Perun system, which are using database tables with no corresponding entity in Perun. It is not possible to use Hibernate for such database operations, yet there are two opportunities for how to solve this issue.

6.1 Integration to Perun

6.1.1 Full JDBC replacement

In this case, the JDBC will be fully replaced by Hibernate. The Issue with not corresponding entities and database tables is necessary to be solved and I have some ideas how it could be done.

1. For each database table which is used in database operations **create appropriate entity** in code. It means that each entity will have to be mapped as well, therefore, I consider annotations mapping more simple in this case.

2. Many database tables are just a combination of two different tables, but Hibernate can't access them. I explain on the specific example. Table - "RESOURCE_ATTR_VALUES" keeps Attributes for each Resource. One row is an association between one Resource and one Attribute identified by their ids. It is possible to create this table via SQL join, but Hibernate will never be able to access this table through a database, because in HQL select clause you have to put Entity. To avoid creating a new Entity it is possible to add a collection of Attributes to Resource in the backend so the Attributes could be accessed directly via Resource.

3. Eventually, it might be done by native queries which allow using native SQL dialect of the database [19]. Unfortunately, I haven't tested this if it also works on an unmapped table from a database.

In case it will work on unmapped tables, it could be the best and the most simple solution for the issue.

6.1.2 Partially replacement

Another possibility is to replace only the most used queries which have an appropriate entity. They will have an obvious impact on database load by using Hibernate. In this case, it is necessary to somehow combine JDBC and Hibernate transactions. *HibernateTransactionManager* seems to be the right choice for applications with a single Hibernate SessionFactory which is exactly our case. It is possible to mix Hibernate and JDBC but DataSource(*setDataSource(javax.sql.DataSource)*) needs to be the same as SessionFactory DataSource. The Perun instances are built to use a single DataSource so this is automatically accomplished. "Application code needs to stick to the same simple Connection lookup pattern as with DataSourceTransactionManager"[20]. This transaction manager is extended by PerunTransactionmanager so it matches the requirements.

This integration could be processed in a single thesis, usage of HibernateTransactionManager and its combination with JDBC queries and also to test Native queries on Hibernate. In my opinion, it will be best to mix the options which I have mentioned. For queries which are not used so often as the others keep JDBC database operations because it is not necessary to have data in cache which are not used too frequently. Implement the rest of most used queries by Hibernate. Database operations on tables which don't have corresponding entities in Perun could be possibly done by hibernate native queries.

7 Conclusion

The main goal of this thesis was to experimentally test an impact of Hibernate and its caching opportunities on database operations in the Perun system against JDBC usage. After I have read necessary documentation about this ORM tool the queries analysis has been provided. Then I have configured Hibernate into Perun system and I have done the mapping for objects for this purpose and configured the second-level cache. I have implemented a few of most used queries by Hibernate way.

After successful implementation I have tested and measured time duration of whole queries twice, first one with Hibernate and then with *JDBC* implementation. *ORM mapping tool* with caching has shown as faster in testing conditions, so this experimental testing has reached the expected result.

I consider this thesis successful. Besides that I have increased my knowledge about Hibernate, work with *relational database* and *Java* as well, the Hibernate shows at very potential for the Perun system and its database operations. After this positive results I can recommend integration of Hibernate, of course, it is necessary, in such a big project, to thoroughly think considering a code structure and database format as I describe in the sixth chapter.

Bibliography

1. ŠŤAVA, Michal. *Robustní a škálovatelný datový sklad pro systém Perun*. 2015. Available from DOI: 1721.1/11173. Master's thesis. Masaryk University, Faculty of Informatics, Brno.
2. GUTIERREZ, Alvaro J. Towards Better Digital Identity Management: What is Digital Identity? [online]. 2006 [visited on 2018-12-09]. Available from: http://zoo.cs.yale.edu/classes/cs457/spr06/info_paper.pdf.
3. PROCHÁZKA, Michal. *News from the EGI community: id and access management* [online]. Inspired, 2014 [visited on 2018-12-07]. Available from: https://www.egi.eu/wp-content/uploads/2016/08/Inspired-issue-16.pdf?utm_source=.
4. PROCHÁZKA, Michal; LICEHAMMER Slávek; MATYSKA Luděk. Perun — Modern approach for user and service management [online]. 2014 [visited on 2018-12-07]. Available from DOI: 10.1109/ISTAFRICA.2014.6880654.
5. *Java SE Technologies - Database* [online]. Oracle, 1995–2018 [visited on 2018-12-07]. Available from: https://www.oracle.com/technetwork/java/javase/jdbc/index.html.
6. KEITH, Mike; SCHNICARIOL, Merrick. *Pro JPA 2: Mastering the Java Persistence API*. 1st ed. New York: Apress, 2009. ISBN 978-1-4302-1957-6.
7. *Hibernate Getting Started Guide* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: http://docs.jboss.org/hibernate/orm/5.3/quickstart/html_single/#preface.
8. *JDBC Architecture Overview* [online]. eTutorials [visited on 2018-12-10]. Available from: http://etutorials.org/SQL/Postgresql/Part+II+Programming+with+PostgreSQL/Chapter+13.+Using+PostgreSQL+from+a+Java+Client+Application/JDBC+Architecture+Overview/.
9. *Hibernate ORM Final User Guide: Architecture* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#architecture.

10. *org.hibernate Interface SessionFactory* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/SessionFactory.html.

11. FOWLER, Martin. *Patterns of enterprise application architecture: Unit of Work*. 1st ed. Boston: Addison-Wesley, 2003. ISBN 978-0321127426.

12. *org.hibernate Interface Session* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/SessionFactory.html.

13. *org.hibernate Interface Transaction* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/Transaction.html.

14. BAUER, Christian; KING, Gavin. *Hibernate in action: Transactions, concurrency and caching*. 1st ed. Greenwich, CT: Manning, 2005. ISBN 978-1-4302-1957-6.

15. *Getting started guide: Tutorial Using Native Hibernate APIs and hbm.xml Mapping* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: http://docs.jboss.org/hibernate/orm/5.3/quickstart/html_single/#tutorial-native.

16. *Interface AttributeConverter<X,Y>* [online]. Oracle, 1995–2018 [visited on 2018-12-07]. Available from: https://docs.oracle.com/javaee/7/api/javax/persistence/AttributeConverter.html.

17. *The Second Level Cache: Improving performance* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: https://docs.jboss.org/hibernate/stable/core.old/reference/en/html/performance-cache.html.

18. *Getting started guide: The mapping file* [online]. JBoss, 2004–2018 [visited on 2018-12-07]. Available from: http://docs.jboss.org/hibernate/orm/5.3/quickstart/html_single/#hibernate-gsg-tutorial-basic-mapping.

19. *Native Query* [online]. JBoss, 2004–2018 [visited on 2018-12-10]. Available from: https://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/query_native.html.

20. *HibernateTransactionManager* [online]. Spring, 2004–2018 [visited on 2018-12-11]. Available from: `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/orm/hibernate5/HibernateTransactionManager.html`.