Master’s thesis

Application Access to Persistent Ontologies

Bc. Martin Ledvinka

Supervisor: Ing. Petr Křemen, Ph.D.

9th May 2013
Acknowledgements

I would like to thank my parents for their unceasing support and love. I would also like to thank my sister for her valuable advices. Last, but not least, I would like to thank my supervisor, Ing. Petr Křemen, Ph.D., for his enthusiasm, help and this awesome topic.
I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague on 9th May 2013
Czech Technical University in Prague
Faculty of Information Technology
© 2013 Martin Ledvinka. All rights reserved.
This thesis is a school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis
Abstract

Ontologies have become a fully competitive knowledge storing systems. However, programmatic access to them lacks standardization common to relational databases. This thesis proposes a software layer capable of unifying access to different ontology storages. In cooperation with persistence provider implementation JOPA, developed at the Czech Technical University in Prague, it should significantly simplify development of applications using ontologies. As a benefit it should provide efficient concurrent access for multiple clients and to multiple ontology storages.

**Keywords**  Application, Concurrency, Java, Modularization, Ontology, Persistence, Storage, Standard.

Abstrakt

Ontologie se staly plně konkurenceschopnými systémy pro ukládání znalostí. Aplikační přístup k nim bohužel postrádá standardizaci běžnou pro relační
databáze. Tato práce představuje návrh softwarové vrstvy schopné sjednotit přístup k různým ontologickým úložištím. Ve spolupráci s aplikační vrstvou JOPA, vyvíjenou na ČVUT v Praze, by měla výrazně zjednodušit vývoj systémů založených na ontologích. Navíc by měla poskytovat efektivní přístup více klientů k několika ontologickým úložištím současně.

**Klíčová slova** Aplikace, Současný přístup, Java, Modularizace, Persistence, Úložiště, Standard.
Contents

Motivation ............................................................................................................. 1

Ontologies .............................................................................................................. 1
Objectives of this Work .......................................................................................... 1

1 Introduction .......................................................................................................... 3

1.1 Standards ......................................................................................................... 4

1.1.1 Resource Description Framework .............................................................. 4
1.1.2 OWL Web Ontology Language .................................................................. 6
1.1.3 OWL 2 Web Ontology Language ............................................................... 7

1.2 Ontology Modularization .................................................................................. 10

1.3 Existing Storage Access Frameworks ............................................................... 11

1.3.1 Jena ............................................................................................................. 12
1.3.2 OWL API .................................................................................................. 12
1.3.3 Sesame ....................................................................................................... 12

1.4 Existing Ontology Storages .............................................................................. 13

1.4.1 Sesame ....................................................................................................... 13
1.4.2 Jena ............................................................................................................. 14
1.4.3 OWLDB .................................................................................................... 14
1.4.4 OWLIM ..................................................................................................... 14
1.4.5 Virtuoso Universal Server ......................................................................... 14
1.4.6 StarDog ....................................................................................................... 15

1.5 Other Ontology Manipulation Approaches ...................................................... 15

1.6 Java Ontology Persistence API ......................................................................... 16

1.6.1 Why JOPA ................................................................................................ 16
1.6.2 JOPA and JPA ............................................................................................ 17
1.6.3 Transactions ............................................................................................... 17
# Analysis and State of the Art

## 2.1 Comparison of Ontology Access Frameworks

## 2.2 Analysis of Existing Storages

### 2.2.1 Expressiveness

### 2.2.2 Reasoning

### 2.2.3 Concurrent Access and Transactions

### 2.2.4 Summary

## 2.3 Selected Solutions

## 2.4 Existing Ontology Modularization Techniques

### 2.4.1 Theoretical Background of Module Extraction

### 2.4.2 Existing Algorithms for Module Extraction

## 2.5 The OntoDriver Concept

### 2.5.1 Disadvantages of Existing Solutions

### 2.5.2 OntoDriver

# Design

## 3.1 OntoDriver and its Relation to Other Java Technologies

### 3.1.1 Java Database Connectivity

## 3.2 Ontology Oriented Design of OntoDriver

### 3.2.1 Object-Ontological Mapping

### 3.2.2 Ontology Module Extraction

### 3.2.3 Module Extraction Principles

## 3.3 OntoDriver API

### 3.3.1 DataSource

### 3.3.2 Connection

### 3.3.3 Statement and PreparedStatement

### 3.3.4 ResultSet

### 3.3.5 Modifications of the JOPA API

## 3.4 Internal Structure of OntoDriver

### 3.4.1 OntoDriver

### 3.4.2 StorageManager

### 3.4.3 StorageModule

### 3.4.4 StorageConnector

### 3.4.5 Coupling between OntoDriver and Persistence Provider

## 3.4.6 OntoDriver and JOPA Cooperation

# Implementation

## 4.1 Ontology Management and Operation Execution

### 4.1.1 DataSource and Connections

### 4.1.2 OntoDriver

### 4.1.3 StorageModule
4.2 Solutions of Interest ........................................ 53
  4.2.1 Factories .............................................. 53
  4.2.2 Design Patterns ..................................... 54
  4.2.3 Ontology Module Extractor .......................... 56
4.3 Modifications of the JOPA Implementation .............. 57

5 Evaluation ...................................................... 59
  5.1 Tests ....................................................... 59
    5.1.1 Unit Tests .......................................... 60
    5.1.2 Integration Tests ................................... 60
  5.2 Storage Benchmark ...................................... 62
    5.2.1 Entity Persist Benchmark ............................ 62
  5.3 Modularization Benchmark ............................... 64
    5.3.1 Single Context Benchmark I ......................... 65
    5.3.2 Single Context Benchmark II ......................... 66
    5.3.3 Multiple Context Benchmark ......................... 67
  5.4 Benchmark Summary ..................................... 67

Conclusion ....................................................... 69

Bibliography .................................................... 71

A Acronyms ....................................................... 75

B Contents of enclosed CD .................................. 77
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>OWL 2 Structure, taken from <a href="http://www.w3.org/TR/owl2-overview/">http://www.w3.org/TR/owl2-overview/</a></td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Class diagram of the public API of OntoDriver.</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Overall structure of the OntoDriver implementation. Note that this diagram omits some methods for the sake of brevity and readability.</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Sequence diagram of an entity retrieval. The notes at the top show JOPA and OntoDriver demarcation. Some method calls are omitted for the lack of space.</td>
<td>48</td>
</tr>
<tr>
<td>3.4</td>
<td>Activity diagram of the transactional system originally proposed for JOPA.</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>An overview of currently implemented storage module and connector factories</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>Class diagram of entity model of the benchmarks. Methods are omitted for the sake of brevity.</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Comparison of selected features of Sesame, Jena and OWL API. 21
2.2 Feature comparison of selected storages. The numbers in rows represent the following features: (1) **Reasoning expressiveness** – denotes expressiveness of the reasoner used with the specified storage. Note that although Jena does theoretically support external OWL 2 DL reasoners (e.g. Pellet), the storage itself does not support OWL 2 expressiveness. (2) **API expressiveness** – expressiveness of the storage’s native API. (3) **Concurrency and transaction support** – support for concurrency and transactions. (4) **Jena/OWL API** – support for Jena or OWL API access. (5) **Reasoning** – where reasoning with storage native reasoner takes place, either in memory or on storage level. (6) **Query** – SPARQL or SPARQL-DL query support, either by built-in query engine or through external reasoners. 24

5.1 Unit test code coverage of OntoDriver implementation. 60
5.2 Integration test code coverage of OntoDriver implementation. 62
5.3 Comparison of various storages on entity persist. Persisted 1100 entities. All storage modules are OWL API based, the ontology is then transferred to Jena (if necessary) and persisted into the storage. **Total time** represents total time of the benchmark including storage connector initialization, **Persist time** represents duration of the persist cycle, with an already initialized storage connector. Values are averages from ten measurements. 63
5.4 Signature of modules extracted by the modularization benchmarks. *ub* is a name space for http://swat.cse.lehigh.edu/onto/univ-bench.owl#. 65
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Comparison of OWL API connectors without and with module extraction. Twenty concurrently running threads over one ontology context. 1178 entities retrieved. Total time represents total time including thread and storage connectors initialization, Retrieval time represents duration of the retrieval cycle of all threads, including module extraction, but with already initialized storage connector. Heap usage represents MB of heap allocated by JOPA and OntoDriver at the end of the test, just before closing the resources. ASS stands for Average Snapshot Size. Values are averages from ten measurements.</td>
<td>65</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of OWL API connectors without and with module extraction. Twenty concurrently running threads over one ontology context. Ontology consistency check performed. Total time represents total time including thread and storage connectors initialization, Check time represents duration of the consistency check of all threads, including module extraction, but with already initialized storage connector. ASS stands for Average Snapshot Size. Values are averages from ten measurements.</td>
<td>66</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparison of OWL API connectors without and with module extraction. Retrieving 649 entities from 14 contexts. Total time represents total time including connectors initialization, Retrieval time represents duration of the retrieval cycles, including module extraction, but with already initialized storage connector. ASS stands for Average Snapshot Size. Values are averages from ten measurements.</td>
<td>67</td>
</tr>
</tbody>
</table>
Motivation

Ontologies

Ontologies, as a knowledge representation paradigm, have become an important part of today’s IT world. They have been recognized to be for many problems (e.g. in bio-informatics) more appropriate than relational databases. One of the main reasons for this is that ontologies allow knowledge to be captured in a declarative semantically precise way. This way of storing knowledge by relating it to other concepts is in many aspects similar to the human memory. As experience has shown us it is often good to seek inspiration for technical science in nature.

Of course, there are cons to the ontological approach. An ontology can hardly compete in efficiency and speed of answering simple queries with relational database with its rigid structure. The flexibility of formal ontological descriptors is a double-edged blade which can make programmatic access difficult, especially in statically typed programming languages like Java and C++. But it is the power of reasoning, deducing inferences for complex queries, where ontologies are incomparable.

Objectives of this Work

As was stated above there is a problem in programmatic access to data in ontologies. Contemporary programming languages require a stable object model and data tailored to this model, while ontologies adjust to the data which is stored in them. In exchange for object model rigidity, programming languages in collaboration with relational databases provide features like business transactions, multi user access and standardized API, all polished
Motivation

by years of development. There are very few advanced ontology storages which provide at least a portion of the rich RDBMS capabilities.

There are languages for defining ontologies (will be discussed later), which correspond roughly to DDL of relational databases, but as most databases have their own API, also ontology storages have their own public API. In Java and the world of relational databases, this problem is solved by the JDBC standard [14]. Ontologies do not have such a standard, although there exist several widely used implementations (also will be discussed later). As working directly with JDBC and data from it can be somewhat cumbersome, the Java platform defines the JPA 2 standard ([19]), which enables an object oriented way of accessing data from database. Again, there is no such standard nor any commonly used implementation for ontologies. Developers at the Department of Cybernetics at the Faculty of Electrical Engineering of the Czech Technical University in Prague have designed a framework which may overcome this deficiency. It is called JOPA.

It is the goal of this work to create an API which would enable unified access to multiple ontology storage systems and would be pluggable into JOPA. Another objective of this work is to provide an implementation of the new API. This implementation should be modular in that it should allow support for new storage systems to be added easily, it should support transactional processing, taking into consideration transactional system built in JOPA. The third objective of this thesis is to explore the possibilities of ontology module extraction in connection with the designed storage connector. Verifying suitability of the designed API and correctness of its implementation on real world ontologies is the last objective of this work.
One of the most cited definitions of ontology says that ontology is a formal, explicit specification of conceptualization [6].

Since ontology in essence formally describes some knowledge we need a way of representing this description. The standards maintaining World Wide Web Consortium has proposed a language called Resource Description Framework (RDF). It is used widely but as ontologies contain even greater variety of data, the language describing it has to evolve as well. Therefore, the standard of OWL Web Ontology Language (OWL in short) has been defined, over years expanding to version 2. In Section 1.1 I give a brief introduction into these standards.

OWL 2 is a very expressive language but this expressiveness has a disadvantage in that there do not exist efficient storages for it at the present time (in contrast to RDF and OWL) and reasoning has to be done in memory. This problem can be solved by building ontologies from smaller modules. These modules can be linked together by the means of importing. Sometimes, on the other hand, the problem is opposite. We may have an ontology containing a large amount of data and we need a portion of this data to reason over. To enable this, modules can be extracted from an existing ontology, thus creating new ontologies entailing a part of the original ontology. The concepts of ontology modularization and importing are discussed in Section 1.2.

In Section 1.3 I describe the foremost storage access frameworks. Section 1.4 then gives an overview of existing storage systems, beginning with the most primitive one – storing ontology in a file on disk. To complete the ontology access overview I present in Section 1.5 information about other ontology manipulation approaches.
1. Introduction

Finally, in Section 1.6 a description of the Java Ontology Persistence API is provided. This framework is developed at the Czech Technical University in Prague and it tries to bring Java standards known from object relational world to object ontological world.

1.1 Standards

1.1.1 Resource Description Framework

The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web [21].

The restriction that RDF describes information about World Wide Web resources comes from the fact that each resource has to have some kind of identification. The Web provides this kind of identification, it is called Uniform Resource Identifier (URI) and the whole concept of Web is built around it. However, URI being a Web concept does not mean that the identified resource has to come from the Web. It actually does not have to be related to computers at all. Anything that needs to be referred to in a statement can have a URI. If we apply the fact that URI can identify any concept to the definition of RDF, we can generalize it by stating that RDF is a language for representing information about resources that can be identified on the Web, even when they cannot be directly retrieved on the Web [21].

RDF is based on a notion that a resource identified by URI can be described in terms of properties and property values. A statement about some resource may say for example that:

http://www.wikipedia.org has a founder whose name is Jimmy Wales

This statement emphasizes some of the words to illustrate the three key concepts in RDF:

- Subject – the thing the statement describes, in our case http://www.wikipedia.org,
- Predicate – a property of the subject, in our case founder,
- Object – the value of the property, in our case Jimmy Wales.

The statement above is easy to read for a human, but it can give a hard time to computer since the structure of such statements is not always
A *triple* comprising of subject, predicate and object is on the other hand easily processable by a computer. An RDF triple (in the Turtle notation) corresponding to the previous example might look like this:

```
<http://www.wikipedia.org> ex:founder
```

These triples are the basic of RDF. In RDF, all three parts of a triple should be identifiable by URI. To make this easier, RDF contains several pre-defined predicates, such as `rdf:type`, `rdf:value`, `rdf:first`.

RDF can be written in several syntaxes, the most common are RDF/XML, Notation3 (N3), Turtle and N-Triple. An example of the Turtle syntax can be seen in Listing 1.1.

Listing 1.1: Turtle syntax example.

```turtle
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix info: <http://www.university-data.com/elements> .

dc:publisher <http://en.wikipedia.org> ;
info:students 23185 ; #xsd:integer
info:staff 1777 . #xsd:integer
```

### 1.1.1.1 RDF Schema

RDF Schema (RDFS) can be used to define new RDF vocabularies which describe domain specific classes and properties. RDFS thus defines several additional concepts that allow users to create such vocabularies. An example can be `rdfs:Class` which is used to define various classes of objects.

```
ex:Vehicle rdf:type rdfs:Class .
```

This example defines a new class named `ex:Vehicle`. We can then use it to define instances of such a class this way:

```
ex:Car rdf:type ex:Vehicle .
ex:Truck rdf:type ex:Vehicle .
```

---

1The term RDF Triple is a common name for statements consisting of subject, predicate (also known as property) and object in RDF.
1. Introduction

We declared subjects \textit{ex:Car} and \textit{ex:Truck} to be of type \textit{ex:Vehicle}. The \textit{ex} prefix denotes a name space\footnote{Name space is a container for a set of (usually related) identifiers. It allows distinguishing the same identifiers from different name spaces.} \url{http://www.example.org/ontologies/jopa}. I will be using this prefix in examples throughout this thesis.

1.1.2 OWL Web Ontology Language

The OWL Web Ontology Language (OWL for short) is a next step in defining a formal language for machine processing of ontology data. It contains all the concepts of RDF and RDFS and adds new concepts that enable machine reasoning to be effective and complete. Among the new concepts are those which can restrict cardinality of relationships (number of instances at each side of the relationship), equality constructs (e. g. \textit{equivalentClass}, \textit{sameAs}), property restrictions (e. g. \textit{someValuesFrom}) and many more.

There were several reasons why OWL was designed to replace RDF as ontology description language. Among them are limited expressivity of RDF, minimal support for ontology interoperability in RDF, no inconsistency detection in RDF(S). More reasons are described in the OWL Web Ontology Language Reference\footnote{Reasoner is a software which reasons over an ontology, i. e. it answers queries based on data in the ontology.} \cite{20}. OWL also introduced 3 profiles which offer different levels of expressiveness and thus provide the user or implementer with a choice based on his requirements on expressiveness and reasoning performance.

1.1.2.1 OWL Full

OWL Full offers complete expressiveness as defined in OWL language at the price of reasoning completeness. OWL Full allows some features that violate the constraints of description logic reasoners and thus there is no guarantee (and it is not likely) that any reasoner\footnote{Reasoner is a software which reasons over an ontology, i. e. it answers queries based on data in the ontology.} can offer complete reasoning over all features of OWL Full. To give an example of the constraints relaxation, let me quote the OWL Reference:

> It is perfectly legal in OWL Full to have a ”Fokker-100” identifier which acts both as a class name (denoting the set of Fokker-100 air planes flying around the world) and as an individual name (e.g., an instance of the class AirplaneType).\cite{20}
1.1.2.2 OWL DL

In OWL DL the DL acronym stands for Description Logic and it is therefore obvious what this language is based on. OWL DL places a number of constraints on the language constructs and is thus a subset of OWL Full. This subset is the maximal subset against which the research at the time of OWL creation could assure that a decidable reasoning procedure could exist for an OWL Reasoner [20]. If we use the example from the previous paragraph, we can say that a "Fokker-100" can never act as a class name and an individual name at the same time. Another constraint is that object properties and datatype properties are disjoint sets. For more information about these constraints, refer to [20].

1.1.2.3 OWL Lite

OWL Lite is a subset of OWL DL, it abides by all the restrictions defined in OWL DL and adds several more (e.g. forbidden usage of owl:unionOf, owl:complementOf etc.). The idea is to provide a minimal useful subset of the OWL language so that efficient OWL Lite complete reasoner can exist. The reference states that OWL Lite supports the basics for subclass hierarchy construction, containing subclasses and property restriction [20].

1.1.3 OWL 2 Web Ontology Language

OWL 2 Web Ontology Language (OWL 2) is an extension of the OWL language. It adds several features which widen its expressiveness, for example qualified cardinality restrictions, asymmetric, reflexive, and disjoint properties, defines three new profiles (will be discussed later), adds new syntax - OWL 2 Manchester syntax [24]. The structure of OWL 2 can be seen in Figure 1.1 which shows that there are two sides of an OWL 2 ontology - the syntactic side, which defines multiple ways of outputting the ontology, and its semantic side, which is divided into two semantic approaches. OWL 2 has two very similar profiles which in a lot of literature blend in with each other. Those are OWL 2 Full and OWL 2 DL.

The difference between OWL 2 Full and OWL 2 DL is in the way meaning is assigned to ontologies. There are two different ways (as shown in Figure 1.1):

- RDF-Based Semantics – it is an extension of RDFS semantics and is based on viewing OWL 2 ontologies as RDF graphs,
- Direct Semantics – it is based on description logic view of ontologies.
1. Introduction

It is obvious that OWL 2 DL corresponds to the Direct Semantics and OWL 2 Full to The RDF-Based Semantics. However, since OWL 2 Full is undecidable [24], it is OWL 2 DL which is used widely. The reason is simple, it is possible to write a reasoner which for a query returns “yes or no” for OWL 2 DL, but not for OWL 2 Full. Despite the restrictions for OWL 2 DL, there are many things which are forbidden in OWL DL and yet are possible in OWL 2 DL. For example, it is possible for a resource to act as a class name and an individual name as well. Thus, our “Fokker-100” can be a class name and also an individual. In OWL, this would mean that the ontology containing these statements is in OWL Full, in OWL 2 it is OWL 2 DL.

Besides the aforementioned profiles OWL 2 defines three additional profiles, each being based on a specific field of applications for which it should be suitable. These profiles, as is stated in the OWL 2 Primer [24], were identified mostly by their user community size. In contrast to the profile
hierarchy in OWL, none of the three profiles in OWL 2 is a subset of each other. They are all subsets of OWL 2 DL suitable for specific usage fields. The profiles are:

- OWL 2 EL
- OWL 2 QL
- OWL 2 RL

1.1.3.1 OWL 2 EL

OWL 2 EL is designed especially for large biohealth ontologies, such as Galen\textsuperscript{4} or SNOMED-CT\textsuperscript{5}. These ontologies are characterized by complex structural descriptions, huge number of classes and generally a vast amount of data. OWL 2 EL forbids negation, disjunction, role inversion and universal quantification on properties. Therefore, there is no way to say that for example ownerOf and propertyOf are inverses of each other.

OWL 2 EL is especially suitable for domains with structurally complex objects.

1.1.3.2 OWL 2 QL

OWL 2 QL is designed for easy realization through relational database technology. It uses features from RDFS and adds only a few OWL 2 concepts. On the other hand, it restricts for example class axioms in that constructs can be used as subclasses that cannot be used as superclasses. It also disallows property chain axioms and equality, so two properties cannot be defined to be equal.

1.1.3.3 OWL 2 RL

OWL 2 RL is a subset mostly meant for effective reasoning without a great loss of expressiveness. The OWL 2 Primer\textsuperscript{24} defines OWL 2 RL to be suitable for enriching existing RDF data. OWL 2 RL restricts class axioms asymmetrically similar to OWL 2 QL, so constructs can be defined as subclass that cannot be used as superclass.

\textsuperscript{4}Available at \url{http://www.opengalen.org/}. Accessed 2013-04-08.

\textsuperscript{5}Available at \url{http://www.ihtsdo.org/snomed-ct/}. Accessed 2013-04-08.


1. Introduction

1.2 Ontology Modularization

Ontologies represent a knowledge source which can contain a lot of data. It is often not desirable to put all data needed by some application into one giant ontology, since portions of this data can be used separately in another field. Therefore it is much more favourable to keep ontologies in smaller chunks that can be easily transported over the internet and using data from some ontology does not mean importing a giant bulk of useless data which, moreover, slows the reasoning down. Ontology modularization is thus a way of engineering ontologies in a way of developing small sets that can later form a large modular ontology.

There are two main approaches to the ontology modularization, one is to create new ontologies smaller and reuse them by importing. The other is to extract modules from existing ontologies, such modules can then be saved as new ontologies or can be used temporarily to speed up reasoning. The concept of linking ontologies together is well known and has existed since RDF. Module extraction on the other hand is an unexplored territory, there is a lot of research to be done in this field and it concentrates on OWL (and OWL 2) based ontologies.

In the following paragraphs I will give a brief introduction into both approaches to ontology modularization.

1.2.0.4 RDF Named Graphs

RDF is built around the principle of a graph. The net or web I was talking about in the introduction is in another word a graph. Therefore, an RDF triple set is in fact a graph. If we give this graph a URI, we define a named graph\(^2\), which is a pair \((n_g, g)\), where

- \(n_g\) is name of the graph (its URI),
- \(g\) is the set of RDF triples.

A nameless RDF graph is like a nameless ontology, it cannot be referenced anywhere and is virtually useless. By creating a named graph, we give it a referable URI which can be used to import the named graph into another ontology. A named graph can import an arbitrary number of other named graphs and thus concentrate knowledge from numerous resources.

The notion of named graphs transforms to OWL ontologies\(^6\) inherently. Every OWL ontology is expected to have a URI which identifies it and it

\(^6\)Throughout the rest of this thesis I will use the term OWL ontology for both OWL and OWL 2 ontologies. If there is distinction necessary I will distinguish the languages by using their respective names.
1.3. Existing Storage Access Frameworks

Storage access frameworks serve as bridges between the serialized form of an ontology (written to a storage using one of the syntaxes discussed above) and a model of the ontology in some running program. Serialized ontology, e.g. in RDF/XML syntax, is nothing else than an XML document (as long as we store it in a document, more alternatives are discussed in the next section). By transforming it into a model we actually recreate the web of knowledge which the ontology represents, the model is the description of the ontology in RDF, OWL or OWL 2. This model can be then queried, modified and again serialized into a storage. There are three most advanced and widely used frameworks for working with ontologies:

- Jena
- OWL API
- Sesame
1. Introduction

1.3.1 Jena

Jena is a Java framework for working with ontologies. Its development started in 2000 and since 2010 it has been maintained by the Apache Software Foundation. It supports RDF graphs and OWL ontologies, contains rule based inference engine for reasoning with ontologies, query engine for SPARQL queries and also several implementations of RDF stores. It also provides several other tools for working with ontologies and building Semantic Web applications.

Jena has its own reasoner, however it has an API for using third party reasoners as well. The biggest disadvantage of Jena is that it does not support OWL 2, its primary focus is on RDF.

1.3.2 OWL API

OWL API is a Java API and reference implementation for working with OWL ontologies. Its main focus is on OWL 2. It has been developed at the University of Manchester since 2003. In comparison to Jena OWL API does not work with triples, it uses high level axioms which are used in the OWL 2 specification and working with them is arguably less error prone than working with triples.

OWL API has reached version three, each version tracking the evolution of OWL (and OWL 2) itself. It has been adopted by numerous tools, including Protégé and the Pellet reasoner to name a few. Instead of implementing its own reasoner OWL API defines interface OWLReasoner through which various reasoners can be plugged into the OWL API. OWL API’s focus on OWL 2 makes it a promising technology for the future.

1.3.3 Sesame

Sesame is an RDF storage and also a Java API for working with ontologies. Since Sesame is one of the most advanced RDF storages, its API is widely used. The Sesame API is also supported by some of the most advanced ontology storages, including OWLIM and StarDog.

---

8SPARQL is a query language similar to SQL but designed for querying ontologies.
1.4 Existing Ontology Storages

To exploit ontologies one needs a language to describe it and an efficient storage system to store it. I have already given an overview of the languages for describing ontologies, in this section I would like to present some of the ways of storing ontologies. Comparison of these storages is the objective of Section 2.2 in the next chapter.

Ontologies have a disadvantage against relational databases – it is their rapidly changes structure. A relational database has a model, this model contains several tables, each of them having several columns of predefined types. These structures and unlikely to change and if they do change, the changes are only minor (at least they should be). In ontologies on the other hand there is no such thing as model. Each individual can have arbitrary number of properties. Each individual can belong to several classes and have multiple connections to other individuals. Thus it is very difficult to develop a performance-efficient storage for ontologies and, as was stated in the introduction, ontology storages can hardly compete with relational databases in answering simple queries, e. g. finding all individuals with the specified value of some attribute.

The most primitive way of storing ontology data is saving it into a file. Both OWL API and Jena support this kind of storage and it is the most straightforward way of working with ontologies. As long as there is not concurrent access involved and the ontology is relatively small it is an efficient and simple way of storing ontologies. More sophisticated storage systems include RDF triple stores and OWL ontology storages.

1.4.1 Sesame

Sesame is an RDF data storage framework. It does not contain its own persistent storage, instead it can be deployed over a variety of storage systems, including relational databases, keyword indexers or in memory storages. Sesame thus maps RDF data to structures supported by the underlying storage. Sesame, as was mentioned earlier, has its own public API and supports neither Jena nor OWL API access. There exist efforts to develop an adapter of Jena for Sesame, one of them is for example Jena Sesame Model.

13 The term triple store refers to the fact that the storage is designed for RDF triples.  
1. Introduction

1.4.2 Jena

As was stated above, Jena also provides an RDF and OWL storage. It actually provides two types of storages:

- Jena TDB - Jena’s own persistent storage, designed for single machine uses,
- Jena SDB - RDF triple store backed by a relational database.

According to The Berlin SPARQL Benchmark [1], both Jena storages exhibit faster data loading and better scalability than Sesame, Sesame on the other hand has significantly better query performance.

1.4.3 OWLDB

OWLDB[15] is a storage system backed by an object-relational database and using the OWL API. It provides support for OWL 2 DL but its problem is that it is no longer developed. However, it represents one of the few alternatives of ontology storages accessible by OWL API.

1.4.4 OWLIM

OWLIM[16] is a full-blown semantic repository developed by Ontotext. It exists in several versions, ranging from a free of charge OWLIM-Lite version with limited features to OWLIM-Enterprise, a replication cluster with load balancing and automatic fail-over.

OWLIM is one of the most scalable existing storages [12], it provides API for Jena and supports RDF, OWL 2 QL and OWL 2 RL semantics.

1.4.5 Virtuoso Universal Server

Virtuoso Universal Server[17] is a multi purpose enterprise server with RDF storage support. Virtuoso provides relational data management, RDF and XML data management, application server, web services deployment etc. An open-source version of Virtuoso is also available at SourceForge[18].

Virtuoso contains a reasoner which supports a subset of OWL and a SPARQL query engine. Virtuoso is a very sophisticated server with lots of

---

possibilities, RDF storage being only one of them. Therefore I have chosen not to include it in the following chapters. It is described here merely to show that ontologies have truly found their way into the world of enterprise information systems.

1.4.6 StarDog

StarDog\(^{19}\) is a RDF storage with OWL 2 reasoning support developed by Clark and Parsia. It uses the Pellet reasoner, which is also developed by Clark and Parsia, for OWL 2 reasoning and also supports SPARQL queries. StarDog has connectors for Sesame and Jena but its primary access is done using its own public API. A community version with limited resources is provided free of charge.

1.5 Other Ontology Manipulation Approaches

To complete the overview of approaches to ontology knowledge usage this section discusses some other ways of working with ontologies related to the previous sections. To create ontologies one does not always need to build a domain specific application. Ontology engineers can use for this task a variety of ontology editors. These editors are often based on technologies described above, they access an ontology storage and manipulate the knowledge with some ontology handling framework. However, some of them use their own implementations representing ontologies in the programming language model. Ontology editors are useful for general manipulation with triples or axioms, they can visualize relations between concepts an hierarchies of classes, thus making the ontology engineering easier. An example of such application is Protégé, developed at the Stanford University. But ontology editors are not suitable for working with ontologies in domain specific applications, where the knowledge extracted from ontology is used by applications’ business logic.

Another approach to ontology knowledge is not meant for ontology manipulation but it is built upon ontologies. To this realm belong mostly web applications which serve as means of accessing information stored in ontologies and possibly visualizing them. These applications are the base

1. Introduction

of Semantic Web\footnote{More information about Semantic Web can be found for example at http://www.w3.org/2001/sw/. Accessed 2013-04-14.} a movement towards more structured and semantically connected web.

1.6 Java Ontology Persistence API

The Java Ontology Persistence API (JOPA) is a Java framework developed at the Department of Cybernetics at the Faculty of Electrical Engineering of the Czech Technical University in Prague and it servers for mapping between an ontology model and an object oriented model which can be used in standard object oriented programming. It is based on a precisely defined contract between the ontology and the object model \footnote{Homepage at http://www.incunabulum.de/projects/it/owl2java. Accessed 2013-05-05.}. This contract is expressed by means of integrity constraints of three types:

- Compile-time – constraints that can be compiled into the object model,
- Run-time – constraints that cannot be compiled directly into the object model, but can be validated at runtime without serious performance degradation,
- Reasoning-time – constraints that have to be evaluated by reasoning engine.

The integrity constraints are validated by JOPA when a modification of the ontology is issued, thus ensuring valid state of the ontology. This approach is favourable since for instance the compile-time constraints are validated implicitly by semantics of Java and only a small portion of constraints is left for validation by reasoner. This multi level system of integrity constraints and their validation is unique in comparison to other ontology manipulation frameworks. The current implementation of JOPA is build on top of OWL API and uses only file storage.

1.6.1 Why JOPA

The reason JOPA was developed is that current frameworks for manipulation with ontologies are either too low level, like OWL API and Jena, or to high level, like Owl2Java\footnote{Homepage at http://www.incumabulum.de/projects/it/owl2java. Accessed 2013-05-05.}.
The low level approaches are good for developing ontology editors or semantic web search engines, however for using in domain specific applications they are too verbose and code which uses them becomes complex and hard to maintain [10].

The high level approaches on the other hand provide a rather simplistic view of the ontology, which can lead to incorrect assumptions about the ontology. Thus JOPA was designed to stand somewhere in the middle and provide access that would be neither too coarse nor too fine. As an additional benefit it was supposed to be based on existing standards so that it would look familiar to its users (and use best practice techniques under the hood).

### 1.6.2 JOPA and JPA

Therefore, JOPA was designed to adhere as much as possible to the JPA 2 standard. Through instances of `EntityManager` the application can query and manipulate the object model and the requests are seamlessly transformed to the underlying ontology. Although not every feature of the JPA 2 specification have been completely implemented, some of them are not even applicable for the ontology-based object model, JOPA represents a fully capable persistence provider which makes working with ontologies easier and more efficient.

### 1.6.3 Transactions

One of the features of the JPA standard is transactional processing on the object model level. This feature was added into JOPA about two years ago and it was the objective of my bachelor’s thesis. The transactional system in JOPA currently supports only resource local transactions, however it was designed with the JCA [18] and JTA [17] standards in mind [11]. It contains, besides other features, a shared second level cache which is consulted during retrieval requests. The storage connector, whose design and implementation is the objective of this thesis, is expected to cooperate with the transactional system of JOPA.
Analysis and State of the Art

In the previous chapter I gave a brief overview of the technologies I will be dealing with in this thesis. This chapter presents an analysis of the current state of the art of programmatic access to ontologies, beginning with comparison of storage access frameworks in Section 2.1 and analysis of existing storages in Section 2.2. Based on this analysis I present an overview of technologies suitable for this work in Section 2.3. The next section (Section 2.4) elaborates more on the topic of ontology modularization, which is one of the techniques used to achieve goals of this thesis. The last section of this chapter, Section 2.5, discusses shortcomings of the current approaches and proposes a solution in concept of OntoDriver. This concept is based on one side on the JCA view of JPA and JDBC collaboration, where the current version of JOPA represents the JPA part, and on the other side respects the various ontology storage solutions. The OntoDriver serves as a basic idea for the design and implementation presented in the following chapters.

2.1 Comparison of Ontology Access Frameworks

This section presents a comparison of main features of the three ontology access frameworks introduced in the first chapter - Jena, OWL API and Sesame. I will order the frameworks by their expressiveness, starting with Sesame and ending with OWL API.

Sesame is the API of the same named RDF storage. It has a rule based RDF reasoner and does not support third party reasoners by any explicit interface. Therefore neither complete OWL nor complete OWL 2 profile
reasoning is available for Sesame accessed repositories. The next step and somewhat hybrid solution is Jena.

Jena is a framework with support for reasoning in RDF and partially in OWL. Model is the key concept in Jena and it represents the manipulated RDF graph. Other key notions are those representing an RDF triple – resource, property and object. Jena has a built-in rule based reasoner and an API for using other reasoners. It also contains a query engine for processing SPARQL queries. Jena can be used to access ontology storages but has its own storage, too. This storage will be discussed in Section 2.2.

OWL API is tailored to OWL 2 and its developers actually contributed to the OWL 2 standard. It is based on axiomatic approach which handles ontology data on a higher level. OWL API works with ontologies in two ways:

1. It represents an OWL 2 ontology as a set of unmodifiable axioms. These axioms represent a read-only view of the ontology model.

2. Changes are done explicitly via instances of the OWLOntologyChange class. Thus the ontology changes are clearly marked and cannot happen by setting values of existing axioms. The changes are applied through an ontology manager and can be applied in batches. This strategy facilitates support for operations such as undo and redo, as well as optimizations for incremental reasoning.

OWL API contains only a very simple rule based reasoner and relies on a public API through which external reasoners are plugged in. The developers advocate this solution by stating that

By separating reasoning functionality, we can relieve implementers of the burden of this, while allowing those who do provide such implementations to be explicit about this in their advertised functionality. In addition, separation of assertion and inference is important in the implementation of OWL, particularly when developing user applications, as users may need explicit indication as to why, for example, hierarchical relationships are present. [7]

OWL API does not provide any SPARQL engine for two reasons. First, SPARQL is primarily an RDF query language and since OWL API is aimed towards OWL 2, a mutation of SPARQL called SPARQL-DL[22] is

22SPARQL-DL is a subset of SPARQL designed for use with description logics based languages, like OWL DL.
2.2 Analysis of Existing Storages

In this section I would like to compare the ontology storages described in Section 1.4 according to features which are vital for developing a robust ontology managing framework. These features include multi user support, storage level transactions, expressiveness etc. All differences are then compared in Table 2.2.

### 2.2.1 Expressiveness

When talking about storage expressiveness we must take into account two views of this term. There is expressiveness of the stored data and there is expressiveness which is supported by the accessing API and reasoner. I

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Sesame</th>
<th>Jena</th>
<th>OWL API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>RDF</td>
<td>RDF, OWL</td>
<td>OWL 2</td>
</tr>
<tr>
<td>Approach</td>
<td>Triple based</td>
<td>Triple based</td>
<td>Axiomatic</td>
</tr>
<tr>
<td>Reasoning</td>
<td>Built-in</td>
<td>Built-in, through API</td>
<td>Through API</td>
</tr>
<tr>
<td>SPARQL</td>
<td>Built-in</td>
<td>Built-in query engine</td>
<td>Not supported</td>
</tr>
<tr>
<td>Storage</td>
<td>Built-in</td>
<td>File, built-in, third party</td>
<td>File, third party</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of selected features of Sesame, Jena and OWL API.

more appropriate. Second, it again relies on external reasoners to provide such implementation (for example Pellet and its SPARQL-DL engine or OWL2Query\[23\]).

Table 2.1 summarizes some of the differences between Sesame, Jena and OWL API. The main advantage of OWL API is its support for the OWL 2 language, its main disadvantages are no support for processing complex conjunctive ontology queries and only in-memory reasoners. Sesame, on the other hand, can be used in conjunction with e.g. OWLIM, which has its own reasoner which does not need the whole ontology loaded in memory of the client for reasoning. The biggest disadvantage of Jena and Sesame is no support for OWL 2. Since JOPA and my diploma thesis aim to support as many ontology storages as possible, Sesame, Jena and OWL API are expected to be supported by the resulting framework. JOPA is already based on OWL API so one of the goals is to find a way to support Sesame and Jena as well.

---

\[23\] OWL2Query was developed by a team at the Czech Technical University. Its homepage is at [http://krizik.felk.cvut.cz/km/owl2query/index.html](http://krizik.felk.cvut.cz/km/owl2query/index.html). Accessed 2013-05-05.
can illustrate the previous sentence on the example of Sesame. Sesame in fact does support storing OWL data, but since it is accessed primarily by the Sesame API, which is RDF-based, unless there is another layer used in conjunction with it (for instance OWLIM), its maximum expressiveness is RDF. Similarly Jena’s built in reasoners support at most OWL Lite profile, so if we wanted to use OWL DL for storing data in Jena TDB, we would have to plug a third party reasoner into Jena.

There are storages which support selected OWL 2 profiles, OWLIM has an OWL 2 RL and OWL 2 QL reasoner, StarDog supports reasoning over OWL 2 DL. Since OWL API is designed to be used for OWL 2 profiles, OWLDB can be used for storing OWL 2 data as well.

2.2.2 Reasoning

It would be incomplete to mention expressiveness and not describe reasoning capabilities of the ontology storages. These are mostly given by the accessing framework, however there are major differences between storage implementations and these can lead to big performance differences. The key attribute of most of the current reasoners is that they perform reasoning in-memory. This can significantly limit their capabilities because for huge ontologies (with billions of triples or axioms) in-memory reasoning can become a serious performance issue. I do not want to call this characteristic a problem because, on the other hand, for smaller ontologies this can be a very favourable attribute since there is no time consuming hard drive access when the ontology is in memory. Storages like Jena TDB, Jena SDB or OWLDB are subject to this characteristic, because they use, for OWL (2) reasoning, such reasoners, e. g. Pellet, HermiT, FaCT++, RacerPro, KAON2.

Some commercial storages, in my list namely OWLIM and StarDog, include built-in reasoners which work on the storage level and do not require the whole ontology to be in memory for reasoning for some ontology profiles.

2.2.3 Concurrent Access and Transactions

It is obvious that concurrent access is problematic for ontologies stored in a file on hard disk. The system locks the file whenever it is accessed and since there is no layer above the file which would manage the user connections concurrent access becomes unreliable. Jena TDB uses a write-ahead log based transactions which facilitate concurrent access, Jena SDB uses the underlying database to handle concurrency, Sesame uses internal repository synchronization, OWLIM has its own concurrency managing system which
has two transaction modes, \textit{fast} for quick data saving with no guarantees of
data integrity in case of abnormal termination, and \textit{safe} for safer approach
which writes data to disk at the end of each transaction. Stardog supports
multi-reader, single-writer concurrency. OWLDB does not support any
explicit concurrency management and relies on the underlying database.

Transaction support is closely related to concurrent access since most
applications realize concurrent access through transactions. Besides file
access, which evidently has no internal support for transactions, all the
discussed storages support transactional processing through their internal
transaction management system. Database backed storages implement it in
collaboration with the underlying database, OWLIM has the two types of
transactions mentioned earlier. OWLDB does not support explicit trans-
actions, it hides the database back end from the user and thus behaves
similarly to an OWL API accessed file.

\subsection*{2.2.4 Summary}

Table \ref{table:features_summary} summarizes differences in features of the described storages. It
may seem that OWL API accessed storages are, compared to Jena ac-
cessed storages, primitive, since they do not support neither transactions
nor SPARQL. But there are good reasons for this situations. First is that
RDF storages have much longer time to evolve, OWL 2 specification was re-
leased in late 2009 and storages that support it, e. g. OWLIM and StarDog,
only support chosen profiles of it and only through modifications to their
internal structures. Besides OWLDB, there is currently no native OWL 2
store.

SPARQL support is another justifiable deficiency. SPARQL is primarily
an RDF query language and for description logic it has several problems,
e. g. that RDF representation mixes the syntax of the language with its
assertions. Therefore a modified version of SPARQL – SPARQL-DL, has
been proposed by Sirin and Parsia \cite{16}. Support for this query language
has been built into multiple reasoners, also thanks to my supervisor Petr
Kr\v{r}men, who co-developed the SPARQL-DL query processor for the Pellet
reasoner.

\subsection*{2.3 Selected Solutions}

Providing support for all of the storages and ontology access frameworks
described above would certainly be a daring task that is hardly possible
to accomplish within a diploma thesis. The primary task of this work
2. Analysis and State of the Art

Table 2.2: Feature comparison of selected storages. The numbers in rows represent the following features: (1) **Reasoning expressiveness** – denotes expressiveness of the reasoner used with the specified storage. Note that although Jena does theoretically support external OWL 2 DL reasoners (e.g. Pellet), the storage itself does not support OWL 2 expressiveness. (2) **API expressiveness** – expressiveness of the storage’s native API. (3) **Concurrency and transaction support** – support for concurrency and transactions. (4) **Jena/OWL API** – support for Jena or OWL API access. (5) **Reasoning** - where reasoning with storage native reasoner takes place, either in memory or on storage level. (6) **Query** – SPARQL or SPARQL-DL query support, either by built-in query engine or through external reasoners.

<table>
<thead>
<tr>
<th>Feature</th>
<th>File + Jena</th>
<th>File + OWL API</th>
<th>Sesame</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>OWL-Lite, OWL 2 DL (external)</td>
<td>OWL 2 DL (external)</td>
<td>RDF</td>
</tr>
<tr>
<td>(2)</td>
<td>RDF</td>
<td>OWL 2 Full</td>
<td>RDF</td>
</tr>
<tr>
<td>(3)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>(4)</td>
<td>Jena</td>
<td>OWL API</td>
<td>None</td>
</tr>
<tr>
<td>(5)</td>
<td>In memory</td>
<td>In memory</td>
<td>In memory</td>
</tr>
<tr>
<td>(6)</td>
<td>Built-in</td>
<td>Reasoner</td>
<td>Built-in</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feat.</th>
<th>Jena TDB, SDB</th>
<th>OWLDB</th>
<th>OWLIM</th>
<th>StarDog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>OWL-Lite, OWL 2 DL (external)</td>
<td>OWL 2 DL (external)</td>
<td>OWL 2 RL, QL</td>
<td>OWL 2 DL</td>
</tr>
<tr>
<td>(2)</td>
<td>RDF</td>
<td>OWL 2 DL</td>
<td>OWL 2 RL, QL</td>
<td>OWL 2 DL</td>
</tr>
<tr>
<td>(3)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>(4)</td>
<td>Jena</td>
<td>OWL API</td>
<td>Jena</td>
<td>Jena</td>
</tr>
<tr>
<td>(5)</td>
<td>In memory</td>
<td>In memory</td>
<td>Storage-level</td>
<td>Storage-level</td>
</tr>
<tr>
<td>(6)</td>
<td>Built-in</td>
<td>Reasoner</td>
<td>Built-in</td>
<td>Built-in</td>
</tr>
</tbody>
</table>

is to design a solution which would be easy to extend and to provide an experimental implementation which would also explore the possibilities of module extraction in connection with ontology access. Therefore I had to select a few storages for which support will be implemented first. The selection is based mostly on my and the JOPA developers’ experience with the storages.

The first and most obvious storage system is a simple file. Implementation for it is a good starting point and I can reuse, with some modifications, the solution already implemented in JOPA. Since JOPA is built on top of OWL API and the main focus of JOPA is towards OWL 2 DL, OWLDB is the next solution I used. Now as we want to support the main storage mechanisms the support for Jena accessed storages has to be incorporated
in some way. The first proof of concept is based on a file manipulated by Jena, using Jena SDB and TDB is only a matter of configuration.

To examine at least some of the potential of commercial ontology storages I had the opportunity to work with OWLIM, although only through its Jena connector. Working directly with OWLIM would probably be more interesting since it would enable using OWL 2 QL and OWL 2 RL constructs and the full power of OWLIM’s reasoner, but it would involve implementing mapping from JPA 2 metamodel used by JOPA to OWLIM API and is thus left for future work.

2.4 Existing Ontology Modularization Techniques

Ontology module extraction is an important part of the effort to make ontology access efficient. For example existing bio-informatics ontologies often contain hundreds of thousands of classes and such ontologies become too heavyweight for reuse. The key motivation for module extraction can be explained by a simple example. Suppose that an ontology engineer is developing an ontology describing some company, including its staff and their health. If an employee suffers from some disease, for example *diabetes*, the engineer might want to include some information about this disease so that the employee’s managers can take his illness into consideration when assigning work to him. But the engineer is no medical expert and thus wants to reuse definitions from an existing ontology. But importing a whole bio-informatics ontology means importing thousands of axioms the engineer does not need. Therefore, it would be favourable if he was able to extract only a portion of the medical ontology which is relevant for his needs.

Hence ontology module extraction means extracting a module of relevant information based on some input data. This module should be as small as possible while on the other hand guaranteeing that it contains all the appropriate data.

2.4.1 Theoretical Background of Module Extraction

Module extraction is based on theoretical background defined in description logics, a family of knowledge representation formalisms which underlie modern ontology languages, such as OWL and OWL 2 [5]. The OWL 2 language (and OWL 2 DL as well) is based ([23]) on $\sf{SROIQ}$ description logic [5]. Theory for module extraction is based on $\sf{SHOIQ}$ description logic which is less expressive that $\sf{SROIQ}$ and is the foundation of the
OWL DL language [9]. However, it is still reasonable to address module extraction, since there are very few ontologies which exploit all the possibilities of OWL 2 DL. The theory behind tractable module extraction is based on three key terms:

- **Conservative extension**
- **Locality**
- **Syntactic and Semantic locality based modules**

To describe these terms let me first define several other concepts, namely **signature**, **interpretation** and **model**.

A **signature** (or **vocabulary**) $S$ of a description logic $L$ is the (disjoint) union of countably infinite sets $AC$ of **atomic concepts** representing sets of elements, $AR$ of **atomic roles** representing binary relations between elements, and $Ind$ of **individuals** representing constants [5]. By $\text{Sig}(O)$ we denote signature of an ontology, by $\text{Sig}(\alpha)$ we denote signature of an axiom $\alpha$.

An **interpretation** $I$ is a pair $I = (\Delta_I, \cdot_I)$, where $\Delta_I$ is a non-empty set, called the **domain** of the interpretation, and $\cdot_I$ is the **interpretation function** that assign to every $A \in AC$ a subset $A_I \subseteq \Delta_I$, to every $r \in AR$ a binary relation $r_I \subseteq \Delta_I \times \Delta_I$, and to every $a \in Ind$ an element $a_I \in \Delta_I$ [5].

An interpretation $I$ is a **model** of an ontology $O$ if $I$ satisfies all axioms in $O$ (written $I \models O$). An ontology $O$ **implies** an axiom $\alpha$ (written $O \models \alpha$) if $I \models \alpha$ for every model $I$ of $O$ [5].

### 2.4.1.0.1 Module

Let $O_1 \subseteq O$ be two ontologies and $S$ a signature. We say that $O_1$ is a **$S$-module** in $O$ w.r.t. a language $L$, if for every ontology $P$ and every axiom $\alpha$ expressed in $L$ with $\text{Sig}(P \cup \{\alpha\}) \cap \text{Sig}(O) \subseteq S$, we have $P \cup O \models \alpha$ iff $P \cup O_1 \models \alpha$ [4].

The module definition says that $O_1$ is a module in $O$ if for every ontology $P$ and axiom $\alpha$ which is in signature $S$, $P \cup O$ and $P \cup O_1$ have the same set of logical consequences with respect to signature $S$.

### 2.4.1.0.2 Conservative extensions

To theoretically define the way ontology modules are extracted, we also need to define a notion of **conservative extension**.

Let $O_1 \subseteq O$ be two ontologies, $S$ a signature and $L$ a logic. We say that $O$ is a **deductive $S$-conservative extension** of $O_1$ w.r.t. $L$, if for every axiom $\alpha$ over $L$ with $\text{Sig}(\alpha) \subseteq S$, we have $O \models \alpha$ iff $O_1 \models \alpha$. We say that $O$ is
2.4. Existing Ontology Modularization Techniques

A **model \( S \)-conservative extension** of \( O_1 \) if, for every model \( \mathcal{I}_1 \) of \( O_1 \), there exists a model \( \mathcal{I} \) of \( O \) such that \( \mathcal{I}|_S = \mathcal{I}_1|_S \) [4].

To rephrase the previous definitions in human language, the first one says that if \( O \) is a **deductive \( S \)-conservative extension** of \( O_1 \), it means that the additional axioms in \( O \) do not add new logical consequences over the vocabulary \( S \), i.e. the knowledge about axioms from \( S \) is completely entailed in \( O_1 \). The second definition describes model conservative extension in terms of models and it states a similar thing, if \( O \) is a **model \( S \)-conservative extension** of \( O_1 \) then every model of \( O_1 \) can be expanded to a model of \( O \) without changing the interpretation of symbols from \( S \).

The **model \( S \)-conservative extension** is also called **semantic conservative extension** and the **deductive \( S \)-conservative extension** is also called **syntactic conservative extension**. Semantic conservative extension is strictly stronger and thus every model \( S \)-conservative extension is also a deductive \( S \)-conservative extension. The notion of conservative extension is used when defining locality and ontology modules.

### 2.4.1.0.3 Locality

From the paragraphs above it is obvious that model conservative extension is a sufficient condition for a module. The difficulty is that deciding model conservative extension is an undecidable problem in description logics backing OWL and OWL 2 [4]. But these notions can be used to define an approach which ensures sufficient conditions for a module and is tractable. This approach is based on **locality**.

Let \( S \) be a signature. We say that an axiom \( \alpha \) is **local** w.r.t. \( S \) if every trivial expansion of any \( S \)-interpretation to \( S \cup \text{Sig}(\alpha) \) is a model of \( \alpha \). We denote by \( \text{local}(S) \) the collection of all axioms that are local w.r.t. \( S \). An ontology \( O \) is **local** w.r.t. \( S \) if \( O \subseteq \text{local}(S) \) [4]. In other words if ontology \( O \) is local w.r.t. \( S \) it means that every interpretation of symbols from \( S \) can be expanded to a model of \( O \) which interprets symbols which are not in \( S \) as an empty set. The locality defined here is also called **semantic locality** and modules based on this locality are called **locality based \( S \)-modules**.

Testing for semantic locality is theoretically a difficult problem (NEXPTIME-complete for \( SHOIQ \) DL), but modern DL reasoners are optimized for standard reasoning tasks and are expected to perform reasonably well [4]. For description logics up to \( SHOIQ \) there exists another notion called **syntactic locality**.

Let \( S \) be a signature. We define two sets of concepts for a signature \( S \) as follows:

\[
\text{Con}^\perp(S) = A^\perp |\neg C^\perp | C \sqcap C^\perp | C^\perp \sqcap C | \exists R.C^\perp | (\geq nR^\perp.C) | (\geq nR.C^\perp)
\]

\[
\text{Con}^\top(S) = \top |\neg C^\perp | C^\perp \sqcap C^\top | C^\top \sqcap C^\perp
\]

\[27\]
2. Analysis and State of the Art

Where ⊥ is an empty concept, also called the bottom concept, ⊤ represents all concept names, also known as the top concept, ¬ is complement of concepts, ∩ is intersection of concepts, ∃ is existential restriction and ≥ defines cardinality restriction. Given these symbols, the definition uses $A^\perp \notin S$ as an atomic concept, $R$ as a role and $C$ as a concept, $C^\perp \in \text{Con}^\perp(S)$, $C_{\oplus}^\top \in \text{Con}^\top(S)$, $i = 1, 2$, and $R^\top \notin \text{Rol}(S)$ as a role. An axiom $\alpha$ is syntactically local w.r.t. $S$ if it is one of the following forms:

1. $R^\perp \sqsubseteq R$
2. $\text{Trans}(r^\perp)$
3. $\text{Funct}(R^\perp)$
4. $C^\perp \sqsubseteq C$
5. $C \sqsubseteq C^\top$
6. $a : C^\top$

A $\text{SHOIQ}$-ontology $O$ is syntactically local w.r.t. $S$ if it is a subset of a set of all axioms syntactically local w.r.t. $S$.24 Algorithms for extracting syntactic locality based modules are polynomial w.r.t. the size of ontology and signature, thus being less expensive than semantic locality based algorithms, and they exhibit similar module quality. They do not extract minimal possible modules but the extracted modules are sufficiently small.24

2.4.2 Existing Algorithms for Module Extraction

The current ontology module extraction research is mostly aimed at locality based modules. The algorithm described in [15] is patterned on the theoretical basics elaborated in [5] and [4] and produces syntactic locality based modules. A version of this algorithm is implemented in OWL API.

There are also other approaches to module extraction, for example the PROMPTFACTOR algorithm, part of the PROMPT suite25 which extracts ontology modules by determining transitive closure of all relations, including subclass-of relation but excluding superclass-of relation, of the specified terms. This way it extracts a module completely entailing knowledge relevant to the specified terms [13]. However, according to benchmarks

24The syntax in the cited paper is slightly different but the $\top$ and $\perp$ concepts are more common in other literature.
25The PROMPT suite is part of the Protégé ontology editor.
2.5 The OntoDriver Concept

This section points out disadvantages present in the current ontology access solutions and proposes a new layer which unifies access to different ontology storages and manages an efficient access to them.

2.5.1 Disadvantages of Existing Solutions

The current situation in ontology manipulation is a little labyrinthine. There are numerous vendors who usually provide a complete application stack, from a storage to a framework for working with the ontology and often up to a web application or an ontology editor. Every such vendor uses his own API, sometimes with additional support for other widely used APIs (such as Jena). Reasoner implementers usually have to provide API for multiple such frameworks, e.g. Pellet has connectors for Jena, OWL API and Protégé. Multiple problems can arise from such a situation.

2.5.1.1 Level and Ease of Use

Section 1.6 already described a dilemma between the low level ontology access APIs like OWL API and Jena and the high level ontology manipulation frameworks like Owl2Java. This conflict can be solved using JOPA. An illustration of the difference between JOPA and OWL API is a simple read of an individual of some class. In JOPA, such class can be represented by an entity class defined in the object metamodel and reading an individual with all properties associated with it is a matter of a single call to the EntityManager. In OWL API, this means reading all data properties and object properties manually. Another example can be seen in Listing 2.1. Here we declare Mary to be John’s wife first using JOPA and then using OWL API. It is obvious that even in this trivial example JOPA saves us

---

in [3] the syntactic locality based algorithm produces significantly smaller modules. Furthermore the PROMPTFACTOR algorithm does not support OWL 2 profiles.

Another approach is the one implemented by OntoFox, it uses a SPARQL-based module extractor which works with RDF ontologies. Effectively the algorithm takes the union of the concise bounded description of the terms [22]. However, the modules extracted by OntoFox do not correspond to the modules defined above since they are RDF based.
three lines of code and is much more readable, maintainable and less error prone.

Listing 2.1: JOPA versus OWL API example. Entity lookup and changes saving is omitted for the sake of brevity.

```java
// JOPA applies the changes to the transactional ontology without persisting them
johnE.setWife(maryE);

// AND NOW THE SAME USING OWL API

// We need the object property
OWLObjectProperty hasWife = factory.
    getOWLObjectProperty( IRI.create(ontologyIRI + "#hasWife"));

// Then we create an axiom
OWLObjectPropertyAssertionAxiom axiom =
    factory.getOWLObjectPropertyAssertionAxiom( hasWife, johnInd, maryInd);

// AddAxiom is the explicit change we are performing
AddAxiom addAxiom = new AddAxiom(ont, axiom);

// And we apply this change to the ontology in memory
manager.applyChange(addAxiom);
```

2.5.1.2 Multiple Storages Access

Another problem, which is currently not solvable using only JOPA, is using multiple ontology storages at one time. In the present we would have to maintain access to all the storages separately, although OWL API and Jena do support multiple ontologies loaded at one time. But when there are multiple storages with different access, for instance one accessed through OWL API, another through Jena and an OWLIM storage accessed through its API, the code becomes difficult to handle and maintain, since each storage requires different code for data manipulation and each storage has to be managed separately.

Nevertheless, accessing multiple storages simultaneously can be a very beneficial strategy. Besides the fact that there are existing knowledge sources, for example as part of the Linked Data\footnote{Linked Data consists of interrelated datasets across the web and is the base of Semantic Web. More information can be found at 
\url{http://www.w3.org/standards/semanticweb/data}. Accessed 2013-05-05.}, whence we can gather information relevant to some domain, we can also use the different expressiveness and performance of various storages. We may store part of the ontology which does not require advanced reasoning in a fast RDF storage...
2.5. The OntoDriver Concept

and at the same time store the rest of the ontology, containing more so-
nified data, in a file and use some OWL 2 DL reasoner for reasoning
over it. Together the two storages form a fast storage system with high
expressiveness.

2.5.1.3 Transactions and Concurrent Access

Keeping the idea of using multiple storages at the same time, not all of
them may support transactional access, and if they do, these transactions
have to be handled individually. Some storages, especially those for OWL
API, do not support any level of access concurrency and changes are visible
immediately after they are applied. The only way how to avoid other users
seeing incomplete changes is to create a collection of changes and apply them
an once. This workaround however does not solve the missing rollback
functionality.

JOPA handles transactions on the business logic level [11], but there
is a need for transactions on storage level to make sure for example that
changes are propagated atomically to the storage.

2.5.1.4 Performance

It is common that multiple users may query for similar data. In case of stor-
ages like OWLDB, Jena TDB or Jena SDB this means that every time an
individual with its properties is requested the underlying storage is queried.
This is highly inefficient. JOPA contains a second level cache [11], which
speeds up entity lookup. More serious performance issues can be caused
by concurrent access to a Jena or OWL API managed storage. Since these
frameworks load ontologies into the main memory it is easily conceivable
what can happen when multiple users access the ontology on the same
machine but through multiple instances of OWL API.

2.5.2 OntoDriver

Based on the analysis presented in this chapter and influenced by the stan-
dards for the Java platform, I propose a new ontology access concept called
OntoDriver. It is inspired by JDBC drivers which are used to unify access
to relational databases. Since JOPA is very similar to the JPA 2 standard,
OntoDriver is designed as a JDBC driver-like layer under JOPA. In co-
operation with JOPA the OntoDriver should solve the issues identified in

\[28\] Rolling changes back is one of the basic concepts of transaction atomicity; all oper-
ations either succeed as a single unit or fail as a single unit.
2. Analysis and State of the Art

this section and thus make development of applications based on ontologies easier and faster.

2.5.2.1 Requirements

The main purpose of OntoDriver is to provide a single API for access to multiple ontology storages managed by different public APIs, including OWL API, Jena and vendor specific APIs. Similarly to JDBC drivers, the OntoDriver should be usable separately as a direct ontology access manager or in collaboration with JOPA, which would serve as the persistence provider (JPA 2) implementation.

The philosophy is that the OntoDriver should be able to manage access to multiple storages simultaneously and provide unified access for client applications, whether through JOPA or directly by using means similar to the java.sql.Connection interface used for relational databases. Multiple applications with different object models should be able to use a single driver at the same time and the driver will be responsible for correct data manipulation. The driver should support resource level transactions so that multiple operations can be grouped into a single atomic one, regardless of business transactions running in the layers above the OntoDriver.

Another goal of the OntoDriver is to make concurrent access to ontologies efficient, especially regarding the problem of memory consumption and performance in multi user environment. A way to achieve this goal may be ontology modularization, whose theoretical background was described in this chapter. My task is not to implement module extracting software but to reuse existing solutions for the driver implementation. Last, but not least, the OntoDriver should have a well defined API so that different implementations may be created.
Analysis presented in the previous chapter showed that there is no standardization in programmatic access to ontologies and that there are other problems regarding working with multiple storages simultaneously and code reuse as well. The chapter was concluded with a short sketch of a software layer called OntoDriver which should solve to large extent the described problems. This chapter elaborates on this concept. It begins with classification of the OntoDriver into the layered application stack on the Java platform (Section 3.1). Section 3.2 then discusses general design decisions concerning the OntoDriver, especially considering relationship with JDBC. Section 3.3 describes the design of API of OntoDriver, taking into account the JDBC standard and capabilities of JOPA as OntoDriver’s primary client. Finally, Section 3.4 presents a design overview of the experimental implementation of the OntoDriver.

3.1 OntoDriver and its Relation to Other Java Technologies

I did an extensive research on Java technologies related to persistent storages and especially transactions in my bachelor’s work, hence if any of the information presented in this section may seem incomplete the curious reader may refer to [11]. The concept of a JDBC-like software layer between JOPA and ontology storage was actually also introduced in my bachelor’s work, however with continuing research on this topic some of the ideas which I had at the beginning became obsolete or have changed. For example the original idea was that each driver would manage only one storage and multiple storages would be handled by JOPA. Also the object-
ontological mapping was supposed to be handled by JOPA. But the main idea is still the same, to design a layer resembling existing Java standards as much as possible with ontological background in mind. Since JOPA tries to implement relevant parts of the JPA 2 standard, currently with support for resource local transactions (JTA transactions intended for future development), it is obvious that the OntoDriver API is designed to resemble to a reasonable extent the JDBC standard.

3.1.1 Java Database Connectivity

Java Database Connectivity (JDBC) is the Java platform standard for accessing relational databases [13], there exist four major versions (standard for the fifth is being prepared for Java 8). The main purpose of JDBC compliant software is to provide connections to the underlying storage. Through this connection data can be created, retrieved, modified and deleted. There are three key concepts which represent the interface of the driver for the client application.

3.1.1.1 Connection

The actual connection to the storage is represented by implementations of the \texttt{java.sql.Connection}\footnote{The classes and interfaces described in this section all belong to packages \texttt{java.sql} or \texttt{javax.sql}. The reader will hopefully forgive me that I do not use fully qualified names of these types. It is cumbersome and hard to read.} interface. This interface is part of the \texttt{java.sql} package and it is thus obvious that it is designed specifically for connecting to SQL queried storages. It does provide several methods which are essential for every storage connection - close, isOpen, commit and rollback. These methods also comprise the core of an interface of the same name designed for accessing my OntoDriver.

3.1.1.2 DriverManager and Driver

In the first version of the JDBC standard the \texttt{DriverManager} and \texttt{Driver} were the means of connecting an application to a storage. The driver manager manages all existing JDBC drivers in the system, drivers implement the \texttt{Driver} interface and register at the \texttt{DriverManager}. Applications then acquire a reference to a \texttt{Driver} from the \texttt{DriverManager} and the driver provides connections to the storage.
3.1.1.3 DataSource

As of JDBC 2.0 the DriverManager and Driver are not the preferred way of obtaining access to the storage. JDBC 2.0 introduced the DataSource interface which does not use any special driver registering service but can be accessed via JNDI\(^{30}\), making the DataSource more portable and transparent. DataSource implementations can provide such improvements as connection pooling and support for distributed transactions \[11\].

I have created a variation of the DataSource interface. There are modifications which prevent the interface to be reused but it would be strange to acquire connections to an ontology storage via the javax.sql.DataSource interface anyway. The modifications will be discussed later. JOPA uses the data source to acquire connections to the OntoDriver, which manages all the storages in use. The data source can be either instantiated directly by JOPA or can be registered in JNDI and requested by JOPA from it.

3.2 Ontology Oriented Design of OntoDriver

Before I define API and internal structure of the OntoDriver, I should describe reasons why the OntoDriver is designed the way it is. These design decisions were made so that the OntoDriver would fulfil the requirements defined at the end of the previous chapter. The most important decision was to move object-ontological mapping from JOPA to the OntoDriver.

3.2.1 Object-Ontological Mapping

The reason why OntoDriver does the object-ontological mapping instead of JOPA is that there are big differences between relational and ontological approach. Let me demonstrate the crucial differences on a simple JPA entity retrieval operation. In standard JPA and JDBC environment, for entity retrieval the persistence provider does an SQL selection query and the driver returns a result set from which the provider reconstructs the entity based on information in the metamodel. This is not possible in case of ontologies.

\(^{30}\)Java Naming and Directory Interface is a service providing Java applications with a unified interface for accessing files and directories. JNDI specification can be found at [http://docs.oracle.com/javase/6/docs/technotes/guides/jndi/reference.html](http://docs.oracle.com/javase/6/docs/technotes/guides/jndi/reference.html), cit. 2013-05-05.
3. Design

First, the persistence provider does not know whether it is accessing ontology managed by OWL API, Jena or other framework. Since there is no standardization in this area the driver might return an array of `OWLAxioms` as well as an array of `RDFNodes`. I could have tried to design a language to which all frameworks can translate their output, but this hits another problem – we are dealing with three languages. Although RDF, OWL and OWL 2 are closely related, there are also big differences, triple based versus axiomatic approach to name at least one.

Second, in ORM world all information about an entity instance is saved in one row of a table (including potential foreign key to other tables), so the entity is retrieved in a single query. When an entity is retrieved by JOPA, its attributes are loaded axiom by axiom through the methods of OWL API. So JOPA cannot issue a single query which would fetch all the data necessary for entity reconstruction, simply because there is no means to do this. Creating a conjunctive query would be unnecessarily complicated given that all existing frameworks provide methods for retrieving resources and property values.

3.2.2 Ontology Module Extraction

Section 2.4 described theoretical background of ontology module extraction. It was concluded with a definition of `syntactic locality` based module extraction, stating that this approach produces modules entailing knowledge about concepts from the given signature. Module extraction is an important part of the OntoDriver concept. Since OntoDriver can hardly lock any part of the ontology in order to maintain transaction isolation, it creates snapshots of ontologies for the clients. This approach can pose serious performance issues if the driver is accessed by a large number of users.

Such problem is likely to appear for big ontologies. It is also likely that the client application metamodel will not cover all the possible concepts from the ontology, because for example medical ontologies have hundreds of thousands of classes. It is therefore reasonable to use only a relevant part of the ontology in order to improve performance and lower computer resource consumption while retaining correctness of data.

---

31 Object-relational Mapping, denotes mapping between object oriented structures of programming languages and relational structures of relational databases, i.e. between class instances and table rows.

32 This is not strictly speaking true, because the persistence provider may fetch necessary data in multiple queries, especially for references to other entities. But we can consider a simplified case where the premise holds.
From the performance point of view, the algorithm for syntactic locality based module extraction has cubic time complexity w. r. t. to the ontology size \[15\], however reasoning over OWL DL is \textsc{NEXPTIME}-complete and reasoning over OWL 2 DL is \textsc{2NEXPTIME}-complete \[25\]. Other profiles are less computationally complex and some offer polynomial reasoning complexity. Nevertheless it is reasonable to believe that module extraction will improve performance even if it itself takes some time to execute, especially for ontologies based on OWL DL and OWL 2 DL. The memory saving is without doubt since the module will take at most the same amount of memory as a complete ontology snapshot would.

### 3.2.3 Module Extraction Principles

Theoretical background of module extraction was given in the previous chapter, its practical application is as follows. The key part of the module extraction process is to define a correct signature, in analysis denoted by \( S \). Since we need to extract only data related to the client application metamodel, we use the entity classes. The entity classes and their fields are annotated with meta data representing the contract between object oriented model and the ontology (as defined by \[10\]). This meta data is based on IRIs of concepts from the referenced ontology and these IRIs are used as \( S \).

The result of the module extraction process is a \textit{syntactic locality} based \( S \)-module \( \mathcal{O}_1 \) of ontology \( \mathcal{O} \) containing all the data relevant to signature \( S \) based on the application metamodel. The module may not be minimal, but it should be reasonably small and is extracted in polynomial time w. r. t. the size of ontology and signature \[1\].

Listing 3.1 shows a very simple entity class annotated with JOPA annotations, its integrity constraints being

\begin{align*}
(1) \quad & \text{ex:entities}\#\text{OWLClassD} \sqsubseteq (\forall \text{ex:attributes}\#\text{hasA} \\
& \quad \text{ex:entities}\#\text{OWLClassA}), \\
(2) \quad & \text{ex:entities}\#\text{OWLClassD} \sqsubseteq (\leq 1\text{ex:attributes}\#\text{hasA}).
\end{align*}

Such entity would produce a module extraction signature

\[ S = \{\text{ex:entities}\#\text{OWLClassD}, \text{ex:attributes}\#\text{hasA}, \text{ex:attributes}\#D-strAtt}\}. \]

Individuals of such class will be included in the module because they have \texttt{rdf:type} set to \texttt{ex:entities}\#\texttt{OWLClassD}. 

37
3. Design

Listing 3.1: A simple entity class used for module extraction.

```java
@OWLClass(iri = "ex:entities#OWLClassD")
public class OWLClassD {

    @Id
    private URI uri;

    @OWLObjectProperty(iri = "ex:attributes#hasA")
    private OWLClassA owlClassA;

    @OWLDataproperty(iri = "ex:attributes#D-strAtt")
    private String stringAttribute;

    // Getters and setters follow...
}
```

The key point about ontology modularization usage in OntoDriver is that it is completely transparent for the client application. The module entails all the knowledge relevant to metamodel used by the application and changes applied to the module have the same consequences as if they were applied to the original ontology (which is what happens on resource level transaction commit).

3.2.3.1 Module Extraction and Conjunctive Queries

The situation is a little more complicated for conjunctive queries, e.g. SPARQL-DL, issued to the ontology. Such queries are not bound to the application entity model and may require data outside the extracted module. Therefore a query analysis has to be performed and if it encounters concepts which are not related to the module signature, the query has to be executed on full snapshot of the original ontology.

3.3 OntoDriver API

This section deals with the public API which every implementation of OntoDriver has to support. It is, again, designed to resemble interfaces from the `java.sql` and `javax.sql` packages. The class diagram in Figure 3.1 shows the public API of OntoDriver including methods. I will briefly describe all these interfaces in the following paragraphs. The last part of this section deals with modifications I had to do to the public API of JOPA in order to fully exploit the new possibilities introduced by the OntoDriver.
3.3.1 DataSource

The `cz.cvut.kbss.ontodriver.DataSource` interface contains only two methods, both serving for obtaining connections to the OntoDriver. The first method takes no parameters, the second one takes an instance of the `cz.cvut.kbss.ontodriver.PersistenceProviderFacade` interface implementation. The `PersistenceProviderFacade` interface, as can be seen in Listing 3.2, defines two methods, one for obtaining an instance of the `cz.cvut.kbss.jopa.model.metamodel.Metamodel`, which contains information about entity classes, and the other to retrieve an entity from the second level object cache managed by the persistence provider.
3. Design

Listing 3.2: The `cz.cvut.kbss.ontodriver.PersistenceProviderFacade` interface. Javadoc comments are omitted for the sake of brevity and can be found in source codes of the OntoDriver API.

```java
public interface PersistenceProviderFacade {
    public Metamodel getMetamodel();

    public <T> T getEntityFromLiveObjectCache(Class<T> cls, Object primaryKey);
}
```

The reference to the persistence provider, i.e. to the JPA 2 implementation, introduces some level of coupling between OntoDriver and the persistence provider. This decision is a trade off between generalization of access to ontology and decoupling of OntoDriver and persistence provider implementations. Section 3.4.5 describes this coupling and reasons for it.

If we compare the OntoDriver version of `DataSource` interface with the `java.sql` version, we can see that the SQL version is aimed directly towards a specific database, while the OntoDriver version returns connections which support all the available storages at once.

### 3.3.2 Connection

The `cz.cvut.kbss.ontodriver.Connection` instances represent connections to the OntoDriver. As I stated earlier, the interface contains key methods for managing life cycle of the connection – `isOpen` and `close` – and for managing resource level transactions – `commit` and `rollback`. Transactions are started automatically by the driver implementation, which corresponds to the JDBC specification [14]. There are also methods for working with query statements, namely `createStatement` and `prepareStatement`, which correspond to similar SQL-based methods in `java.sql.Connection`.

The rest of the methods in the `cz.cvut.kbss.ontodriver.Connection` interface are CRUD operations with different level of `ontology context` awareness. `Ontology context` is a name I use for representing ontologies and named graphs. Each ontology context represents an ontology and its most important part is the identifier of the ontology. For example, if OntoDriver manages connection to three ontology files, each containing a single ontology with a unique logical URI, then these three ontologies are represented by three ontology contexts, their URIs being the logical URIs of the ontologies they represent. The behaviour of CRUD operations with entity context specified, e.g. `find(Class<T>, Object, URI)` is straightforward,
they are executed on the given context. The methods without context, e. g. `find(Class<T>, Object)` behave in the following way:

**Persist** The entity is persisted either into the default context of the connection (see the `setConnectionContext` method), or into the context previously set for the entity by the `setSaveContextFor` method.

**Find** The entity is retrieved either from the default connection context or, if not found there, from the first context in which individual with the specified URI is found. The order in which ontology contexts are searched is expected to represent their priorities.

**Merge** The entity is merged into the context from which it was retrieved.

**Remove** The entity is removed from the context from which it was retrieved.

More specific description can be found in Javadoc of the OntoDriver API. Since it is also possible for an individual to have an object property value from another ontology, there are also methods which support working with ontology contexts on entity attribute level.

The `cz.cvut.kbss.ontodriver.Connection` interface also defines a method for getting a list of available ontology contexts, represented by the `cz.cvut.kbss.ontodriver.Context` class. The order in which contexts are present in the list also defines their priorities for operations like `find` without context URI. These priorities are specified when the OntoDriver is loaded by the order in which information about storages is passed to the driver.

The `Connection` shares very little with its `java.sql` relative. Apart from the methods for managing connection and transaction life cycle and methods for creating statements the OntoDriver version is absolutely different. The reason is quite simple and stems from the fact that object-ontological mapping is done by the OntoDriver. Therefore the `Connection` is used to work directly with entities, which is reflected on the methods comprising this interface.

### 3.3.3 Statement and PreparedStatement

Conjunctive queries, e. g. SPARQL-DL, are in OntoDriver represented by the `Statement` and `PreparedStatement` interfaces, located in the package...
3. Design

cz.cvut.kbss.ontodriver. The Statement interface represents a conjunctive query passed as a single string. The PreparedStatement in OntoDriver extends the Statement interface and adds the possibility of using parameters and eventually providing some level of injection protection.

These interfaces correspond to the existing java.sql.Statement and java.sql.PreparedStatement interfaces only by their names. While the java.sql versions contain a large number of methods, their OntoDriver counterparts are extremely simple. Despite their simplicity, the OntoDriver statements contain all the methods necessary for query execution. If additional methods are needed in the future, they can be added.

3.3.4 ResultSet

Instances of the cz.cvut.kbss.ontodriver.ResultSet interface represent results of queries issued through statements and prepared statements. Again, this interface resembles to some extent the java.sql.ResultSet interface, providing methods for extracting values in various Java types, especially the built-in primitive ones. The result set can be also iterated by the means of using the hasNext and next methods. It is also possible to navigate through the result set by jumping to the beginning or end or by moving relatively to the current position (the relative method).

Almost all methods from this interface have their counterparts in java.sql.ResultSet, while a lot of methods from the SQL version were omitted, especially the update versions. The most important method added to the OntoDriver result set is the registerObserver method, which enables the query results to be fetched asynchronously. This works on the assumption that reasoning over an ontology may take some time and it can be beneficial to provide the client with a portion of results immediately and deliver the rest when it is available.

3.3.5 Modifications of the JOPA API

Changes to the public API of JOPA are merely to reflect that it is now possible to work with multiple ontology contexts. Therefore, in addition to standard JPA versions of find, merge and persist, new methods for using multiple ontology contexts were introduced to the EntityManager interface. These new methods are shown in Listing 3.4.3. The difference between these newly added methods and those already present in EntityManager

---

33 Refer to the well-known SQL injections which are possible also for SPARQL Update queries.
is straightforward, for example using `persist(Object)` persists the entity into ontology with the highest priority loaded by the OntoDriver, while using `persist(Object, URI)` persists the entity into ontology specified by the context URI. There is no need to add ontology context supporting version of `remove`, because `remove` works only with managed entities, whose context is already known.

Listing 3.3: Methods added to the `EntityManager` interface. Javadoc comments are omitted for the sake of brevity.

```java
public interface EntityManager {

    // Existing EntityManager methods...

    public void persist(final Object entity,
                         final URI contextUri);
    public <T> T merge(final T entity, final URI contextUri);
    public <T> T find(final Class<T> entityClass,
                      final Object primaryKey, final URI contextUri);
    public List<Context> getAvailableContexts();
}
```

The other change is in the way JOPA itself is initialized. It has to be provided with information about the storage(s) so that OntoDriver can connect to them. A list of `cz.cvut.kbss.ontodriver.OntologyStorageProperties` is passed to the `EntityManagerFactory` implementation constructor for this reason. These properties contain information about an ontology context, as can be seen in Listing 3.4.

Listing 3.4: Attributes of the `OntologyStorageProperties` class.

```java
public class OntologyStorageProperties {
    /** URI of the ontology */
    private final URI ontologyUri;
    /** URI of the physical storage,
     * e.g. OWLDB database, OWLIM storage, file */
    private final URI physicalUri;
    /** Type of the storage connector. */
    private final OntologyConnectorType connectorType;
    /** User name for the storage, if necessary */
    private final String username;
    /** Password for the storage, if neccessary */
    private final String password;

    // Constructors and getters...
}
```
The connector type specifies whether OWL API or Jena (or possibly other framework) is used for ontology access. The storage properties are currently created by the user application but it is our plan to specify them in a configuration file similar to persistence.xml which is used by standard JPA providers. This file will be parsed by a loader and JOPA will be configured according to parameters specified in it.

3.4 Internal Structure of OntoDriver

In this section I will describe the design of the current implementation of OntoDriver, especially the architecture of the driver itself and software modules which comprise it. Figure 3.2 shows overall structure of the OntoDriver implementation. The central point of the OntoDriver is the OntoDriver interface.

3.4.1 OntoDriver

The OntoDriver implementation manages the whole driver. It provides StorageManagers for the incoming connections and it also keeps a map of factories which are used to create StorageModules and StorageConnectors. The idea is that for each supported connector type (currently OWL API, Jena and OWLIM) the factory class can be specified in properties which are passed to the driver on creation. This way new implementations of modules, connectors and factories can be added freely. When no factory classes are specified, the default ones provided by the driver are used.

3.4.2 StorageManager

The StorageManager interface represents a manager through which connections from the client application (represented by Connection instances) execute operations on the underlying ontologies. It manages a set of modules, represented by the StorageModule interface, which represent ontology contexts. Each ontology context is mapped to a single storage module. The storage manager defines methods for CRUD operations as well as transaction managing methods and when such a method is called, it finds the correct storage module (according to the Context passed to the method) and forwards the request to this module.

\[34\] Strict reader will hopefully forgive me that I use instance in connection with an interface. It is obvious that I mean instance of implementation of that interface.
3.4. Internal Structure of OntoDriver

Each connection is provided with its own storage manager so that concurrently running connection transactions do not interfere with each other.

3.4.3 StorageModule

Each instance of the StorageModule represents a single ontology context. The storage module is responsible for carrying out the operations issued on its ontology context by the client application (through the Connection and StorageManager). The storage module is transactional, transactions beginning at the first CRUD operation launch since the module instantiation or since the last commit or rollback. When a module transaction is started, a snapshot of the underlying ontology context is created and all operations during the transaction are carried out on the snapshot. The transaction...
can then end in two ways:

**commit** On commit the changes that were applied to the snapshot are applied to the main ontology.

**rollback** On rollback the snapshot is simply thrown away and all pending changes are lost.

I have already stated that the storage module takes care of operations on the underlying ontology context. This also means that the storage module performs the object-ontological mapping. Data come into the storage module from the layers above as entities and they are transformed to axioms or statements according to the information in metamodel.

It is obvious that the approach I have taken is far from being similar to typical JDBC and JPA cooperation, where the object-relational mapping is done by the persistence provider. The reasons for this decision concern the scattered nature of data in ontologies and were discussed in Section 3.2.

### 3.4.4 StorageConnector

The `StorageModule` represents a single ontology context and is responsible for working with the ontology, but it is an instance of `StorageConnector` through which the `StorageModule` communicates with the persistent storage. The previous chapter has shown that various storages are accessible through different APIs and the connectors are the means I use to unify access to them.

The connectors are closely related to storage modules. For every ontology context `StorageModule` and `StorageConnector` are created by the same factory, for example when the ontology is accessed through OWL API, the factory produces an OWL API based storage module which uses a connector provided by the same factory. The factory knows what kind of connector to create based on information about the storage passed to the OntoDriver.

### 3.4.5 Coupling between OntoDriver and Persistence Provider

The coupling introduced in the `DataSource` interface has reasons already described in Section 3.2, the main one being the fact that object-ontological mapping is done by the OntoDriver. Therefore the OntoDriver needs access to the application metamodel.
That having said, it is not always necessary to provide the OntoDriver with a facade to the persistence provider. For example when the user application decides to work directly with the driver via Connections and issues only conjunctive queries which do not need object-ontological mapping, it would be unreasonable to require provider facade for this type of usage. Therefore, it is possible to use the driver API without it. In this case a default empty implementation of the PersistenceProviderFacade interface is provided by the driver itself. This solution will work as long as no object-ontological mapping is required.

3.4.6 OntoDriver and JOPA Cooperation

So how do the OntoDriver and JOPA cooperate? The overall mechanism is fairly simple, every UnitOfWork, which represents the persistence context, acquires a connection to the OntoDriver from the shared server session [11]. When an entity is retrieved from the storage, its clone is registered in the connection instead of the original so that modifications to the clone are correctly propagated to the transactional snapshot in the corresponding storage module.

An example of a call sequence from the top-most layer of JOPA to the physical connection in OntoDriver is shown in Figure 3. This figure illustrates an entity instance retrieval, initiated by a client application calling the find method on an EntityManager. This sequence diagram shows only public API method calls and invocations of methods bearing significance to this explanation. The example shows situation when the storage module and connector have not yet been initialized for the specified Connection. Therefore they are first acquired from the DriverFactory and then the entity is retrieved. Since it is the first call after the module initialization a resource level transaction is started and the ontology context is cloned. Were the module already initialized and the transaction already running, the ontology would have been already cloned.

3.4.6.1 Transactions

Transactions are the most important part of the OntoDriver-JOPA cooperation. When the user finishes a business transaction on the EntityManager level, the resource level transaction running in the connection is also finished and changes are either propagated to their respective ontology contexts or rolled back, depending on the call. The transaction life cycle corresponds to the Figure 3.4, which was originally introduced in [10], where the front-end is represented by JOPA and the back-end is represented by the OntoDriver.
3. Design

Figure 3.3: Sequence diagram of an entity retrieval. The notes at the top show JOPA and OntoDriver demarcation. Some method calls are omitted for the lack of space.
Figure 3.4: Activity diagram of the transactional system originally proposed for JOPA.
The previous chapter described, besides the OntoDriver API, the design of my implementation of the OntoDriver. In this chapter I will first present some specific details of implementation of OntoDriver\footnote{From this chapter on I will refer to the OntoDriver implementation as OntoDriver. If the distinction between the OntoDriver implementation, the OntoDriver as a concept and the OntoDriver API is necessary, I will point it out.} (Section 4.1). In Section 4.2 I point out the most interesting solutions I used, including design patterns application. The rest of this chapter deals with modifications I had to make to JOPA (Section 4.3).

4.1 Ontology Management and Operation Execution

The overall structure of the OntoDriver is known to the reader from Section 3.4 and Figure 3.2 from the previous chapter. Now I will describe some of the implementation details.

4.1.1 DataSource and Connections

The current version of the driver uses only one DataSource implementation, the \texttt{cz.cvut.kbss.ontodriver.impl.SimpleDataSource} class. It creates a new connection for every incoming request and does not perform any pooling. There is pooling implemented in the driver, but deeper inside on the physical storage connection level. In contrast to JDBC, OntoDriver connections are lightweight objects since they do not represent the actual connection to the storage, but only connection to the OntoDriver. Since
storage connectors are handled by the OntoDriver, there is little need for connection pooling on the DataSource level.

Every instance of `cz.cvut.kbss.ontodriver.impl.ConnectionImpl`, which represents the connection, maintains a collection of all entities that were retrieved (or persisted) through it. Therefore it is not possible for example to merge changes on an instance of entity retrieved by a different connection. This solution facilitates lazy loading of entity attributes and ontology context tracking. For operations like update and delete the layer using `Connection` does not have to keep track of the context to which an entity belongs, because the connection does it.

The data source contains reference to the central point of the OntoDriver, an instance of the `OntoDriverImpl` class.

### 4.1.2 OntoDriver

The `cz.cvut.kbss.ontodriver.impl.OntoDriverImpl`, besides providing `StorageManager` instances, also manages a collection of factories which provide instances of `StorageModule` and `StorageConnector`. Factories can be registered at the `OntoDriverImpl` by passing the appropriate factory class in properties when the driver is instantiated, otherwise the default ones are used. When a storage module or connector is requested, the `OntoDriverImpl` chooses the correct factory based on the type of the connector to which the factory is mapped. Currently supported connectors are OWL API, Jena and OWLIM.

### 4.1.3 StorageModule

The function of a storage module has been described earlier, so I will concentrate on the fact that there are multiple implementations of the `StorageModule`. These implementation reflect firstly the different supported access frameworks (OWL API, Jena and OWLIM) and secondly multiple types of ontology snapshots. The type of the snapshot depends on the `StorageConnector` implementation. There are generally three kinds of connectors and snapshots:

1. For the basic type every connector connects directly to the storage and the snapshots represent the whole ontology from the storage. This solution is not feasible for concurrent access since at every commit the whole ontology is regenerated by the committing module,

2. Caching type uses a single shared storage connection, every snapshot is created from this connection and its changes are merged back into
the shared connection on commit. The connector uses read/write locking so that multiple reads can occur concurrently.

3. Module extracting connector uses OWL API and its syntactic locality based module extractor. There is a single shared connection which holds the whole ontology and every module is provided with a relevant ontology module, which acts as the transactional snapshot.

In the current version Jena and OWLIM ontologies are internally transformed to OWL API based snapshots, because we have neither Jena nor OWLIM-based object-ontological mapping implemented. Therefore only OWL API based storage modules are used in the current version of OntoDriver. The modules use object-ontological mapping mechanisms which were originally present in JOPA and are based on the formalism proposed in [10]. An overview of existing storage module and connector factories is shown in Figure 4.1. A pair of StorageModule and StorageConnector implementations correspond to each of the factories. But a lot of code is actually reused through the Decorator pattern [3].

4.2 Solutions of Interest

The OntoDriver implementation contains several solutions which I would like to emphasize. The modular nature of the Driver is facilitated by the layered system in which commands are executed. Starting with the StorageManager, which manages all available ontology contexts and distributes incoming requests to the target modules, continuing with the way ontology manipulation in StorageModule is decoupled from the storage connection represented by StorageConnector.

4.2.1 Factories

The simplicity of adding new storage access strategies and support for new frameworks would not be possible were the Abstract Factory design pattern not embedded in the OntoDriver design. However there also has to be a way how to tell the driver which factory to use when there are multiple options. The first design was using a static registration method, which had to be called prior to OntoDriver initialization. This approach has obviously many disadvantages, because there would have to be someone who would call this registration method. I tried using a static initializer block but this block is called only after the class is loaded by the Java class loader.
4. Implementation

- DriverOwlapiFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverJenaCachingFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverOwlapiCachingFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverJenaFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverModularizingOwlapiFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverAbstractFactory
  - `getContexts() : List<Context>`
  - `releaseStorageConnector(StorageConnector) : void`
  - `releaseStorageModule(StorageModule) : void`

- DriverOwlimFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverModularizingOwlapiCachingFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

- DriverOwlapiFactory
  - `createStorageConnector(Context) : StorageConnector`
  - `createStorageModule(Context) : StorageModule`

Figure 4.1: An overview of currently implemented storage module and connector factories.

Therefore I decided to use another strategy. I introduced a configuration parameter which can specify which factory is to be used for a type of connector. This way the user may choose the implementation by just setting another class name in the driver configuration. Similar strategy is used for example in the HotSpot Java virtual machine, where the user can configure which type of garbage collector should be used.

### 4.2.2 Design Patterns

*Design Patterns* are general reusable solutions to common problems and were first described and catalogued in the famous book *Design Patterns: Elements of Reusable Object-Oriented Software* [3]. My work uses several of the design patterns described in this book. I will group these patterns in the same way they are classified in the book, into creational patterns, structural patterns and behavioural patterns.
4.2.2.1 Creational Patterns

The Abstract Factory was mentioned in several places. Another pattern often used in OntoDriver and in JOPA as well is the Builder pattern, which simplifies creation of objects with multiple attributes that can be null.

4.2.2.2 Structural Patterns

By employing the Decorator pattern a high level of code reuse is achieved for some types of storage connectors, especially the caching types are in essence decorating a shared instance of the non-caching type. Decorator is also used for wrappers of several OWLAPI classes, enabling me to set attributes of otherwise immutable classes. This is necessary for the mechanism where transactional changes from the cloned ontology are applied to the original ontology on commit. JOPA uses decorators for collections, where the decorators wrap standard Java collection implementations so that when the collection changes the change can be tracked by the persistence context. The PersistenceProviderFacade is a very simple implementation of the Facade pattern, providing only two methods to the whole persistence provider implementation.

4.2.2.3 Behavioural Patterns

The Template Method pattern is a common way of implementing specific parts of an algorithm in subclasses and it is used at several places in my implementation. The ontology cloning quite resembles the Memento pattern, although the system is not exactly the same as the one described in [3]. The original ontology serves as a memento so that if an undo operation (a rollback) is executed, the working copy is restored to the original state. The ResultSet interface, although not implemented in the current version, is designed to represent an Iterator over the results of a conjunctive query.

The last pattern used in OntoDriver is Strategy. It is widely used by storage connectors, since they usually define one public API for the accessing module, but there are differences in the way they connect to the storage, e. g. OWL API connectors for file and OWLDB or Jena connectors for file and Jena TDB. By using the strategy pattern each family of connectors defines a single API, but the implementation is chosen at runtime based on the type of the storage.
4. Implementation

4.2.3 Ontology Module Extractor

One of the OWL API based connectors uses syntactic locality based module extraction. When an ontology snapshot is requested at the beginning of a resource level transaction, the central connector, which manages the shared ontology, extracts a syntactic locality based module which serves as the transactional ontology snapshot in the storage module. The signature which is used for module extraction is, as was stated earlier, generated from the metamodel of the client application. The module extractor is part of the OWL API implementation and is based on the techniques described in [3] and [5], the algorithm itself being introduced in [15]. The modularizing connector caches existing extraction signatures mapped by their metamodel, so that the signature need not be extracted every time a connector is created.

There are currently two limitations to the way OntoDriver extracts ontology modules. The first is the lack of support for properties, defined by the `cz.cvut.kbss.jopa.model.Properties` annotation. Simply put, the properties defined in a `java.util.Map` annotated with `Properties` specify properties and values which are not part of the contract defined by JOPA, but can be present in individuals of the class represented by the entity. However since JOPA itself does not fully implement support for properties, it is not a serious deficiency of the OntoDriver.

The second limitation is incomplete support for entity types, defined by the `cz.cvut.kbss.jopa.model.Types` annotation. The types denote OWL classes to which an individual belongs. Since the module extractor signature is generated only from the metamodel, without actual knowledge of the instances, it is currently impossible to determine which ontology classes (i.e. types), besides the type defined by the `OWLClass` annotation, should be added to the signature.

These limitations can be in the present overcome by additionally specifying a set of IRIs which should be included in the module extraction signature in properties passed to JOPA on start up. In addition to that JOPA is able to track changes to `Types` annotated fields in entities and add corresponding types to the module signature. In the future it would be more comfortable if JOPA or the OntoDriver were able to use some heuristics to determine what extra data should be added to the signature.
4.3 Modifications of the JOPA Implementation

The changes to the public API of JOPA were already discussed in the Design chapter, Section [3.3]. In this section I would like to describe some of the changes I made to the implementation of JOPA. The most obvious change is that object-ontological mapping has been moved from JOPA to OntoDriver implementation. Similar situation is with the changes which removed the old ontology connection layer and replaced it with a shared StorageAccessor instance, which provides connections to the OntoDriver. The StorageAccessor is managed by the server session and is thus shared by all persistence contexts.

Some changes reflect the fact that from now on JOPA can work with multiple storages at the same time and that multiple concepts with the same URI can exist and not be considered the same as long as they are unique in their respective ontology context. Hence the CacheManager,UnitOfWork and CloneBuilder implementations have been modified accordingly. The CloneBuilder implementation contains another modification which relates to the way Connection registers entities which were retrieved through it. Since the client application actually works with a clone of the registered object, the CloneBuilder has to register the clone in the Connection instance associated with the persistence context so that changes to the clone are propagated through the Connection into the ontology snapshot.

The rest of the modifications of the JOPA implementation are mostly improvements and bug fixes, especially of the transactional system.
This chapter presents an evaluation of the OntoDriver implementation. Since my work is not based on any existing solution\footnote{With the exception of object-ontological mapping from JOPA.}, it is vital to prove that the implementation is correct. This is discussed in Section 5.1. Section 5.2 presents and explains results of a simple benchmark which I created to compare different ontology storages and storage access solutions. Section 5.3 discusses another benchmark which I used to test the module extracting solution. The last section (Section 5.4) presents a short summary of the benchmark results.

5.1 Tests

It is hard to evaluate correctness of a design. Its validity can be proved or disproved only in long term by its adoption and by gathering feedback from developers who use it. However, the OntoDriver design was done as similar as possible to the existing JDBC standard and the requirements stemming from the differences between databases and ontologies were studied carefully in order to minimize the risk of inappropriate design.

What can be tested is the OntoDriver implementation. The tests I did include unit tests, which validate implementation of individual software modules (be it a method or a class), and integration tests, which validate functionality of the driver as a whole. JOPA implementation contained approximately 180 test cases, both unit and integration. Most of these tests I wrote when I was developing the transactional system for JOPA\footnote{With the exception of object-ontological mapping from JOPA.}. Currently JOPA and OntoDriver contain altogether around 370 test cases, so the number of tests has more than doubled.
5. Evaluation

<table>
<thead>
<tr>
<th>Package</th>
<th>Line coverage</th>
<th>Branch coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cz.cvut.kbss.ontodriver.impl</td>
<td>64.73%</td>
<td>46.47%</td>
</tr>
<tr>
<td>cz.cvut.kbss.ontodriver.impl.utils</td>
<td>31.37%</td>
<td>28.57%</td>
</tr>
</tbody>
</table>

Table 5.1: Unit test code coverage of OntoDriver implementation.

5.1.1 Unit Tests

It is always hard to test layers which serialize data to some persistent storage. One of the ways to go is to heavily use stubs and mock objects. This is exactly the approach I used in unit tests for the OntoDriver implementation. Table 5.1 shows unit test code coverage of the OntoDriver implementation. Packages which deal with connectors and object-ontological mapping are not included in this table, because their coverage is too low and are better covered by integration tests. The unit tests mostly cover classes like ConnectionImpl, OntoDriverImpl and StorageManagerImpl in package cz.cvut.kbss.ontodriver.impl.

5.1.2 Integration Tests

Integration tests verify that a larger software module or the whole software system work correctly as a single unit. For the OntoDriver this means both testing the implementation and also its cooperation with JOPA.

The first set of tests operates directly on Connection instances and is used mostly to verify that various storage connectors and storage modules work correctly, including behaviour in exceptional cases. These tests perform operations like entity persist, retrieval, modification and removal. An example of such a test can be seen in Listing 5.1 where the test persists the same entity into several ontology contexts and then verifies that the entities were really persisted.

Listing 5.1: Example of an integration test working directly with Connection.

```java
@Test
public void testPersistIntoAll() throws Exception {
    LOG.config("Test: persist the same 
            + "entity into all contexts.");
    acquireConnection("MixedMultiContextsPersistIntoAll");
    c.setAutoCommit(false);
    final List<Context> contexts = c.getContexts();
    final URI uriB = entityB.getUri();
    for (Context ctx : contexts) {
        c.persist(uriB, entityB, ctx.getUri());
    }
}
```
The second set of tests uses JOPA and simulates behaviour of a client application, which works only with `EntityManager`. Again, these tests use multiple storage connections. For randomly chosen tests I also manually verified that the data were physically saved to ontology files. A short example of an integration test which uses JOPA is shown in Listing 5.2.

Listing 5.2: Example of an integration test working JOPA through the `EntityManager`.

```java
@Test
generic void testRemoveFromRelationship() {
    LOG.config("Test: remove owner of a relationship.");
    em = TestEnvironment.getPersistenceConnector(
        "RemoveRelationshipOwner", storages, true);
    em.getTransaction().begin();
    em.persist(entityD);
    em.persist(entityA);
    em.getTransaction().commit();

    final OWLClassD d = em.find(OWLClassD.class, entityD.getUri());
    assertNotNull(d);
    assertTrue(em.contains(d));
    final OWLClassA a = d.getOwlClassA();
    assertTrue(em.contains(a));
    em.getTransaction().begin();
    em.remove(d);
    assertTrue(em.contains(a));
    assertFalse(em.contains(d));
    em.getTransaction().commit();

    final OWLClassD resD = em.find(OWLClassD.class, entityD.getUri());
    assertNotNull(resD);
    final OWLClassA resA = em.find(OWLClassA.class, entityA.getUri());
    assertNotNull(resA);
    assertEquals(entityA.getStringAttribute(), resA.getStringAttribute());
}
```
5. Evaluation

<table>
<thead>
<tr>
<th>Package</th>
<th>Line coverage</th>
<th>Branch coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cz.cvut.kbss.ontodriver.impl</td>
<td>62.45%</td>
<td>50.00%</td>
</tr>
<tr>
<td>cz.cvut.kbss.ontodriver.impl.jena</td>
<td>70.68%</td>
<td>50.00%</td>
</tr>
<tr>
<td>cz.cvut.kbss.ontodriver.impl.owlapi</td>
<td>66.11%</td>
<td>44.91%</td>
</tr>
<tr>
<td>cz.cvut.kbss.ontodriver.impl.owlapi</td>
<td>66.82%</td>
<td>48.39%</td>
</tr>
<tr>
<td>cz.cvut.kbss.ontodriver.impl.utils</td>
<td>68.63%</td>
<td>60.00%</td>
</tr>
</tbody>
</table>

Table 5.2: Integration test code coverage of OntoDriver implementation.

Integration tests also greatly increase code coverage of the OntoDriver implementation. Table 5.2 shows the coverage. Note that there is a lot of uncovered branches which handle invalid state and arguments, while these arguments are already handled by JOPA. Such branches are not invoked by these integration tests.

5.2 Storage Benchmark

I used ontology generator from the LUBM\textsuperscript{37} ontology benchmark for my ontology benchmarks. The entity model consisted of five simple classes, class diagram of which can be seen in Figure 5.1, which did not represent all the classes available in the ontology because I intended to use the same model for the modularization benchmark (see the next section). Ontologies created by the generator had in average 7000 axioms, ranging from about 8500 axioms to less than 6000. My goal was to do entity retrieval and persist benchmark to show differences between various storages. However, since the current OntoDriver implementation uses OWL API based ontology modules which load the whole ontology into memory, the retrieval benchmark would have a little testifying value. Therefore it is not present here.

5.2.1 Entity Persist Benchmark

The persist benchmark is able to provide some interesting results. Although it also uses an in-memory model of the ontology, it has to store the new instances into the storage. Therefore we are able to compare performance of batch insertion of various storages. Of course, this benchmark does not produce completely unbiased results, since it always replaces the whole original ontology with the new one, while in future for example for OWLIM or Jena TDB the OntoDriver would only insert the newly added or modified values.

5.2. Storage Benchmark

![Class Diagram](image)

Figure 5.1: Class diagram of entity model of the benchmarks. Methods are omitted for the sake of brevity.

<table>
<thead>
<tr>
<th></th>
<th>OWL API + File</th>
<th>OWL API + OWLDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time / s</td>
<td>3.246</td>
<td>141.915</td>
</tr>
<tr>
<td>Persist time / s</td>
<td>0.967</td>
<td>122.674</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Jena + File</th>
<th>Jena + Jena TDB</th>
<th>Jena + OWLIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time / s</td>
<td>4.500</td>
<td>3.692</td>
<td>7.948</td>
</tr>
<tr>
<td>Persist time / s</td>
<td>1.333</td>
<td>1.576</td>
<td>2.338</td>
</tr>
</tbody>
</table>

Table 5.3: Comparison of various storages on entity persist. Persisted 1100 entities. All storage modules are OWL API based, the ontology is then transferred to Jena (if necessary) and persisted into the storage. Total time represents total time of the benchmark including storage connector initialization, Persist time represents duration of the persist cycle, with an already initialized storage connector. Values are averages from ten measurements.

On the other hand, this benchmark at least simulates conditions when a complete in-memory ontology is persisted to a storage. A situation which can emerge for example when an ontology is transferred from a file to an OWLIM repository. The results of the benchmark are shown in Table 5.3.

It is obvious from the results that OWLDB is by far the slowest ontology storage in our comparison. I did some research of the OWLDB structure and also made some slight modifications to improve its performance but the biggest performance penalty comes from extensive use of Hibernate events, especially the PostInsert event. Every time a record is inserted into the database a new transaction is started and does some housekeeping.

---

5. Evaluation

work. The problem is that the PostInsert event is raised for every single insertion, which in our case means several thousands of invocations and short lived transactions because every JOPA entity has multiple fields which correspond to data or object properties and are inserted separately. A much more efficient way would be to perform these tasks in some preliminary part of the commit operation, when it is sure that the data will not be changed any more. However, this change would require major redesign of OWLDB and this is not the goal of my work.

Other storages performed according to expectations, OWLIM would most likely outperform other solutions for higher load and if it did not have to treat the ontology through OWL API and Jena. OWL API storing to a file is the fastest one, but there are two comments necessary. First, it is the only situation, besides OWL API + OWLDB, where there is no intermediate step in the ontology serialization (for example OWL API → Jena → storage). Second, the file does not have any transactional or concurrency management system, which would introduce overhead as in the case of Jena TDB or OWLIM.

5.3 Modularization Benchmark

The modularization benchmark is the most interesting part of my evaluation, because it compares capabilities of ontology module oriented solution with simple ontology snapshots. From the modularization point of view it is of no use trying to compare different storages since they all use an in-memory OWL API model of the ontology anyway. Therefore I restricted the benchmark to thorough comparison of module extraction based and standard OWL API connectors, both using a file as the persistent storage.

The LUBM ontologies and the entity model I used have one disadvantage – they are too simple. The generated ontologies contain each in average 7000 axioms and there are only thirteen classes in them, five of which are covered by my entity model. The signature used for module extraction in these benchmarks is shown in Table 5.4. Moreover, about half of the individuals in the ontologies are those of class UndergraduateStudent which is also in the entity model. Hence the extracted modules cannot be smaller than half of the original ontology. For real life medical ontologies, like SNOMED-CT, the ratio between the extracted module and the original ontology is likely to be much larger.

I created three different benchmarks, two use a single larger ontology accessed by multiple entity managers concurrently. One retrieves about one thousand entities from it, the other uses this context for consistency
5.3. Modularization Benchmark

<table>
<thead>
<tr>
<th>Classes</th>
<th>Object Properties</th>
<th>Data Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ub:University,</td>
<td>ub:advisor,</td>
<td>ub:name,</td>
</tr>
<tr>
<td>ub:UndergraduateStudent</td>
<td>ub:teacherOf,</td>
<td>ub:emailAddress</td>
</tr>
<tr>
<td>ub:Course,</td>
<td>ub:worksFor,</td>
<td></td>
</tr>
<tr>
<td>ub:University,</td>
<td>ub:memberOf,</td>
<td></td>
</tr>
<tr>
<td>ub:GraduateStudent</td>
<td>ub:takesCourse</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Signature of modules extracted by the modularization benchmarks. *ub* is a namespace for http://swat.cse.lehigh.edu/onto/univ-bench.owl#.

<table>
<thead>
<tr>
<th></th>
<th>Without Modularization</th>
<th>With Modularization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time / s</td>
<td>8.960</td>
<td>7.495</td>
</tr>
<tr>
<td>Retrieval time / s</td>
<td>7.252</td>
<td>5.717</td>
</tr>
<tr>
<td>Heap usage / MB</td>
<td>54.240</td>
<td>57.513</td>
</tr>
<tr>
<td>ASS / axioms</td>
<td>17495</td>
<td>11862</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison of OWL API connectors without and with module extraction. Twenty concurrently running threads over one ontology context. 1178 entities retrieved. *Total time* represents total time including thread and storage connectors initialization, *Retrieval time* represents duration of the retrieval cycle of all threads, including module extraction, but with already initialized storage connector. *Heap usage* represents MB of heap allocated by JOPA and OntoDriver at the end of the test, just before closing the resources. *ASS* stands for Average Snapshot Size. Values are averages from ten measurements.

check. The third one uses fourteen smaller contexts and retrieves entities from them through a single entity manager.

### 5.3.1 Single Context Benchmark I

The single context benchmark uses one larger ontology with approximately 17000 axioms. Multiple connections to the OntoDriver are requested by multiple entity managers and they all retrieve more than a thousand entities from the context. The entity managers are all open in concurrently running threads which are all started at the same moment. The main goal is to measure difference in memory consumption and processing time between the connector which uses full ontology clones and the modularizing connector. Results can be seen in Table 5.5. The solution without modularization uses caching connectors, which share a single storage connection.
5. Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Without Modularization</th>
<th>With Modularization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time / s</td>
<td>7.890</td>
<td>5.330</td>
</tr>
<tr>
<td>Check time / s</td>
<td>6.082</td>
<td>3.490</td>
</tr>
<tr>
<td>ASS / axioms</td>
<td>17495</td>
<td>11862</td>
</tr>
</tbody>
</table>

Table 5.6: Comparison of OWL API connectors without and with module extraction. Twenty concurrently running threads over one ontology context. Ontology consistency check performed. Total time represents total time including thread and storage connectors initialization, Check time represents duration of the consistency check of all threads, including module extraction, but with already initialized storage connector. ASS stands for Average Snapshot Size. Values are averages from ten measurements.

Time results show significantly better performance of the module extracting connector. However, the memory statistics show larger memory consumption of the ontology module extracting solution. After thorough investigation I discovered that the set of axioms comprising the extracted module contains significantly less axioms but is bigger in memory than the set of all axioms from the original ontology. Despite my best effort I was not able to find the source of this paradox. It happens during the module extraction in OWL API, but the exact spot remains unknown to me. This result is certainly a great disappointment, because consuming less memory was one of the goals for which ontology module extraction was implemented by OntoDriver.

5.3.2 Single Context Benchmark II

This single context benchmark uses the same setup as the previous benchmark. The difference is that this benchmark uses the reasoner for a consistency check of the underlying ontology. The purpose of this test is to verify that reasoning operations are also (positively) affected by the ontology module extraction. The benchmark uses Pellet as the reasoner. Results are presented in Table 5.6.

The results show that modularization reduces duration of the reasoner operation almost to one half, more precisely to 57.4%. This even exceeds the difference in axiom count, where the extracted module contains 67.8% of axioms of the original ontology.
5.4 Benchmark Summary

<table>
<thead>
<tr>
<th></th>
<th>Without Modularization</th>
<th>With Modularization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time / s</td>
<td>8.657</td>
<td>8.451</td>
</tr>
<tr>
<td>Retrieval time / s</td>
<td>2.068</td>
<td>1.693</td>
</tr>
<tr>
<td>ASS / axioms</td>
<td>7501</td>
<td>5143</td>
</tr>
</tbody>
</table>

Table 5.7: Comparison of OWL API connectors without and with module extraction. Retrieving 649 entities from 14 contexts. Total time represents total time including connectors initialization, Retrieval time represents duration of the retrieval cycles, including module extraction, but with already initialized storage connector. ASS stands for Average Snapshot Size. Values are averages from ten measurements.

5.3.3 Multiple Context Benchmark

This benchmarks loads fourteen ontologies, the biggest one containing 17495 axioms, the smallest one with 5454 axioms. It then searches for about 650 entities, but the entities are from the last two contexts and the test does not provide URI of the context in which the entities should be looked for. Therefore the OntoDriver has to search all the available contexts, ordered by their priority, until it finds the requested entities.

The results of this benchmark are shown in Table 5.7. The difference between module extracting connector and standard connector is not so significant, but again this is partly caused by the fact that the extracted modules contain major parts of the original ontologies (as can be seen in the third row of Table 5.7). Also, the difference between total time and retrieval time is given by the fact that the total time includes initialization of all fourteen storage connectors, which takes some time.

5.4 Benchmark Summary

The benchmarks have revealed some interesting facts. The modularization benchmarks show that module extraction is able to reduce the size of the ontology and speed up working with it, especially the reasoning. On the other hand, the algorithm in OWL API, which is used for the module extraction in OntoDriver, exhibits unusual behaviour causing a smaller set of axioms consume more memory than a larger set of axioms. This is a very unfortunate observation since it prevents achieving the second goal of modularization – to decrease system resources consumption, especially in multi user environment.

It has also turned up that OWL API lacks a competitive, more sophisti-
5. Evaluation

cated persistent storage. OWLDB has proved to be very inefficient in terms of performance. This is bad news for OWL 2 DL ontologies, because OWL API is currently the only widely used framework capable of working with them.
Conclusion

The idea of a JDBC-like software layer unifying access to different storages originates from my bachelor’s work [11]. As the time went, the requirements grew broader around the central idea of a unified ontology storage connector. Currently, I have an implementation capable of working with multiple ontology storages containing distributed data. These data are accessible through the entity model defined by integrity constraints in JOPA. Furthermore the implementation exploits ontology module extraction in order to improve performance of working with OWL ontologies and yet be transparent to the user.

However, there is still a long way to go. The first and most important step is to implement expressive query support and to finish proper transaction isolation implementation, which is currently only half-way. Next, native support for Jena and other APIs is needed. The OWL API solution works, but it can hardly exploit all the possibilities other frameworks offer (in this place I again mention OWLIM and its storage-level reasoning). More work is also need on the JOPA implementation, for example support for persistence.xml-like configuration or entity fields of type java.util.Map, which is not implemented in the current version.
Bibliography


Appendix A

Acronyms

ACID Atomicity Consistency Isolation Durability
API Application Programming Interface
CRUD Create Retrieve Update Delete
DDL Data Definition Language
DL Description Logic
IRI Internationalized Resource Identifier
IT Information Technology
JCA Java Connector Architecture
JCP Jaa Community Process
JDBC Java Database Connectivity
JDK Java Development Kit
JOPA Java Ontology Persistence API
JPA Java Persistence API
JSR Java Specification Request
JTA Java Transaction API
OO Object-oriented
ORM Object-relational Mapping
A. Acronyms

**OWL**  Web Ontology Language

**OWL 2**  Web Ontology Language, Second Edition

**RDBMS**  Relational Database Management System

**RDF**  Resource Description Framework

**RDFS**  Resource Description Framework Schema

**SPARQL**  SPARQL Protocol and RDF Query Language

**SQL**  Structured Query Language

**URI**  Uniform Resource Identifier

**XML**  Extensible Markup Language
Appendix B

Contents of enclosed CD

- readme.txt ..................... the file with CD contents description
- exe ................................. the directory with executables
- src .................................. the directory of source codes
  - jopa-driver ........................ implementation sources
  - thesis ......................... the directory of \LaTeX\ source codes of the thesis
  - text .............................. the thesis text directory
  - DP_Ledvinka_Martin_2013.pdf ..... the thesis text in PDF format