# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Novotný** Jméno: **Michal** Osobní číslo: **420187**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Ontologie ukládání informací v e-commerce prostředí**

Název diplomové práce anglicky:

**Ontology of information storing in e-commerce environment**

Pokyny pro vypracování:

1. Prostudujte state of the art možnosti strojového popisu zboží nabízeného v e-commerce systémech.
2. Navrhněte, implementujte a otestujte systém, který z různých datových zdrojů dokáže shromažďovat data a analyzovat je s cílem sjednocení do logických skupin.
3. Navrhněte jednotný model popisu dat, který umožní nasbíraná data slučovat a následně analyzovat.
4. Navrhněte a implementujte dotazovací mechanismus, který umožní nad nasbíranou analyzovanou množinou dat rozumné dotazování s ohledem na vyhledávací systém.
5. Porovnejte výsledky analýzy a sjednocení dat vzhledem k datům vytvořeným lidmi v existujících e-commerce systémech a dále porovnejte úspěšnost vyhledávání nad sjednocenými daty.

Seznam doporučené literatury:

[1] CONSORTIUM, W. W. W. Resource Description Framework. Available from: <:http: //www.w3.org/2004/OWL>
[2] TIM BERNERS-LEE, J. H. ? LASSILA, O. The Semantic Web. 2001
[3] O?REILLY, T. What is Web 2.0? 2005.
[4] Lops P., de Gemmis M., Semeraro G. (2011) Content-based Recommender Systems: State of the Art and Trends
[5] Adomavicius G., Tuzhilin A. (2015) Context-Aware Recommender Systems

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. Martin Klíma, Ph.D., Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **16.02.2018** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2019**

[redacted]

Ing. Martin Klíma, Ph.D.
podpis vedoucí(ho) práce

[redacted]

podpis vedoucí(ho) ústavu/katedry

[redacted]

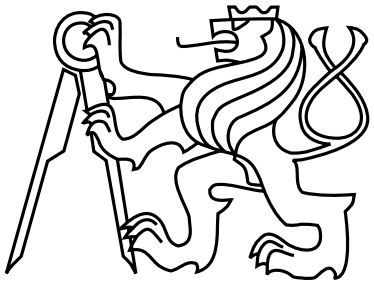
prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání Podpis studenta

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master's Thesis

# Ontology of information storing in e-commerce environment

*Bc. Michal Novotný*

Supervisor: Ing. Martin Klíma, Ph.D.

Study Program: Open informatics, Masters degree

Field of Study: Software Engineering

May 24, 2018

# Aknowledgements

I would like to thank everyone who helped and supported me during this thesis and my whole studying time.

# Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 25, 2018 .....

# Abstract

This thesis is focused on a summary of existing methods and structures for storing e-commerce data. The goal is to design, implement and test a system which collects and analyzes data from multiple e-commerce sources for knowledge extraction. Another part of this theses deals with the design of a unified model for e-commerce data representation which is used to store the extracted semantic data. A method for data querying is designed and implemented to enable logical searching within the data structure. Stored data and extracted knowledge are tested against user annotated dataset. The whole system is implemented as a web application for data search visualization and testing.

# Abstrakt

Tato práce je zaměřená na shrnutí existujících metod a stuktur pro ukládání e-commerce dat. Hlavním cílem je navrhnout, implementovat a otestovat systém, který shromáždí a zanalyzuje e-commerce data z různých zdrojů za účelem extrakce znalostí. Dalším aspektem této práce je návrh jednotného modelu pro popis e-commerce dat. Tento model je využit pro uložení vytěžených sémantických dat. Dále je navrhnuta a implementována metoda pro logické dotazování nad uloženými daty. Uložená data a vytěžené znalosti jsou otestovány proti uživatelsky anotovanému souboru dat. Celý systém je dále naimplementován jako webová aplikace pro vizualizaci uložených dat a testování vyhledávání nad daty.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In today's ever faster-growing Internet environment are e-commerce systems one of the most used systems online. E-commerce can be anything from e-shops and on-line markets to auction sites - anything where people can buy or sell products or services using an electronic system and network. According to 2017 reports from [2] e-commerce is growing exponentially in Europe. Furthermore from 2013 onwards is Czech Republic European country where e-commerce delivers the biggest contribution to enterprises' total revenue - almost quarter of country's total turnover is generated using e-commerce [19].

Since e-commerce is fast growing industry [2], as with every Internet service, shops and e-commerce websites need to handle increasing amounts of data. Not only are there more products and services to offer, but also more and more people rely on these systems to shop and that is really important for marketing strategies and the overall profit of companies in the business.

## 1.1 Problem definition and motivation

E-shops or any other systems are really easy to set-up nowadays and the e-commerce industry is not only for big companies anymore. There are literally hundreds of systems which can provide a company or an individual with online store and many of these systems face the same issues. Further, we will discuss e-shops and products mainly, but the same concepts and problems can be found in any e-commerce environment offering products or services.

One of these issues is how to handle the amount of data generated by the users. This problem has been addressed a lot in the past. But there are more issues - how to use data in a specific way that can help the business thrive. Other problems are tightly coupled to the systems themselves. How to insert, edit and describe product data in e-commerce system in a way that is suitable for searches and other important activities like product categorization or recommendation? A lot of e-commerce systems rely heavily on users to input data like the connection of products with their accessories or product association with their variants or other products.

### 1.1.1 Data meaning and problems

With Web 2.0 and further [27] the meaning of data and the ability to store and retrieve it in a logical way is really important not only in the e-commerce environment. To offer the right product or service, to enable sensible search and recommendation systems and to ensure a better experience for both sides using the system, data need to be structured in a logical way and have to carry some additional information. And this is an issue faced by many businesses. Almost all of the e-shops offer recommendation systems and advanced search engines based on user data (orders, page viewing,...) and product data. But what if these data do not exist or there is not enough of it? What if there is no knowledge of the data meaning in the e-shop?

A lot of systems use *data feeds* which is a shop, product and category and often stock information in a structured format like XML (eXtensible Markup Language) or JSON (JavaScript Object Notation). Data feeds can also be exported from e-shop systems for example in a format for rating and comparing engines. These *data feeds* offer great flexibility for the systems as users do not have to input the data manually. But there are issues as well. Mainly concerning the structures of the feeds which are not unified across the domain. Feeds also do not usually contain all the information about the products and categories, rather a condensation of main attributes such as *name*, *category* and *price*.

## 1.2 Goals of this project

The goal of this project is to analyze the current situation of data storage and representation in e-commerce systems and further to design and implement a system which would create semantic data from existing data and structure them in clear and unified format in the form of a *data feed*. This *data feed* can be used to automatically incorporate data into the system and to enable searching, recommendation and product connection (as *accessory* or *alternative*) on the system level without user's need of input. The semantic data could be created by a user, exported or scraped from existing systems or transformed from original *data feeds*.

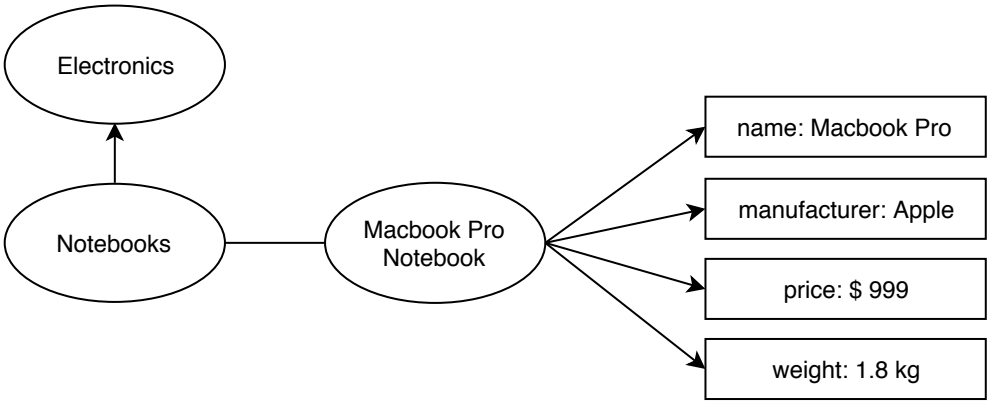



Figure 1.1: E-commerce data top level view

Products often come with attributes or parameters like *manufacturer, price* or *weight* as seen on diagram 1.1. These parameters and their values need to be interpreted correctly if we want to unify the data structure and add meaning to the values. in order to solve this, a system needs to be able to clean the data and transform them into the unified format. Another requirement of the system is to filter any data or values which are not necessary. This system should also be able to learn from data and enable for better and more correct data transformation based on the dataset.

Other parts of the system must be able to process the semantic data and store them. During this process of *consolidation* and storing, data can be clustered and divided into logical groups which will enable better search and other operations.

Requirements of the system and goals of this thesis are summed in the list bellow.

- Data scraping from existing e-commerce sites
- Transformation of data to a unified format
- Consolidation of data, logical grouping and product association
- Storing of the data
- Searching over the stored data

### 1.2.1 Scenarios

In this section are listed and described basic scenarios which the system should support. These operations should make use of the semantically enhanced data and the logical grouping to retrieve specific results.

**I. Product knowledge extraction** System should be able to perform data analysis and use predefined dictionaries or functions to extract meaning from the product descriptions. Numeric values with units should be paired in order to enable unit conversion and unified representation. Extracted data should be stored as knowledge.

**II. Matching similar products** We should be able to create product relationships based on the attributes and meaning they have. We can simply match similar property values like dimensions, price or even categories and retrieve similar products.

**III. General search** A system should have a querying abilities to retrieve stored semantic data in a similar way a normal e-commerce search system would. Additionally it should use the knowledge stored within the data to return more complex results.

**V. Product accessory search** Retrieving product and it's accessories or related products based on the attribute values of the products. The system should use the knowledge to find and connect product as an item and accessory.

# Chapter 2

# Analysis of the problem

In this chapter, we will look into existing ways of storing and describing e-commerce data. We will discuss the meaning of the data and how ontologies are used in today's systems to enhance search and other operations. We will analyze existing tools for data mining and knowledge extraction as well as tools for building ontologies and storing semantic data.

## 2.1 Data in e commerce system

For data analysis we had multiple instances of *data feeds* in XML format as well as access to databases of multiple e-commerce systems. Feeds are summed in table 2.1. Data can be generally divided in two groups: *structured* and *unstructured*. Typical e-commerce system usually consist of *products* (or services) and *categories*. We analyzed multiple domains described in the list below.

| Name | Description | Source |
|---|---|---|
| JCPenny product feed | Over 10000 clothing products in CSV format | https://www.kaggle.com |
| 4Camping product feed | Over 17000 products of clothing and outdoor equipment in Google RSS 2.0[15] format | http://www.cj.com/ |
| Alza.cz data | 12000 products: notebooks, phones and their accessories as HTML files | scraped (see section 2.5) |
| Kasa.cz data | 6500 products: notebooks, phones and their accessories as HTML files | scraped (see section 2.5) |

Table 2.1: Data available for analysis

**Analyzed domains**

- Clothing
- Outdoor equipment (backpacks, camping equipment,...)
- Notebooks

- Notebook accessories (keyboard, mouse,...)
- Phones
- Phone accessories (cases, charging cables,...)

**Structured data** Product and category information is a good example of structured data. Categories are usually hierarchically organized in a multiple tree structure. In figure 2.1 can be seen a classic e-shop top-level view on product and category relationship. Products can be connected to multiple categories and to every category on the way to one of the root categories. Products are represented usually as a set of features which we call *attributes*. Some of these features may be present in e-shop data feeds.

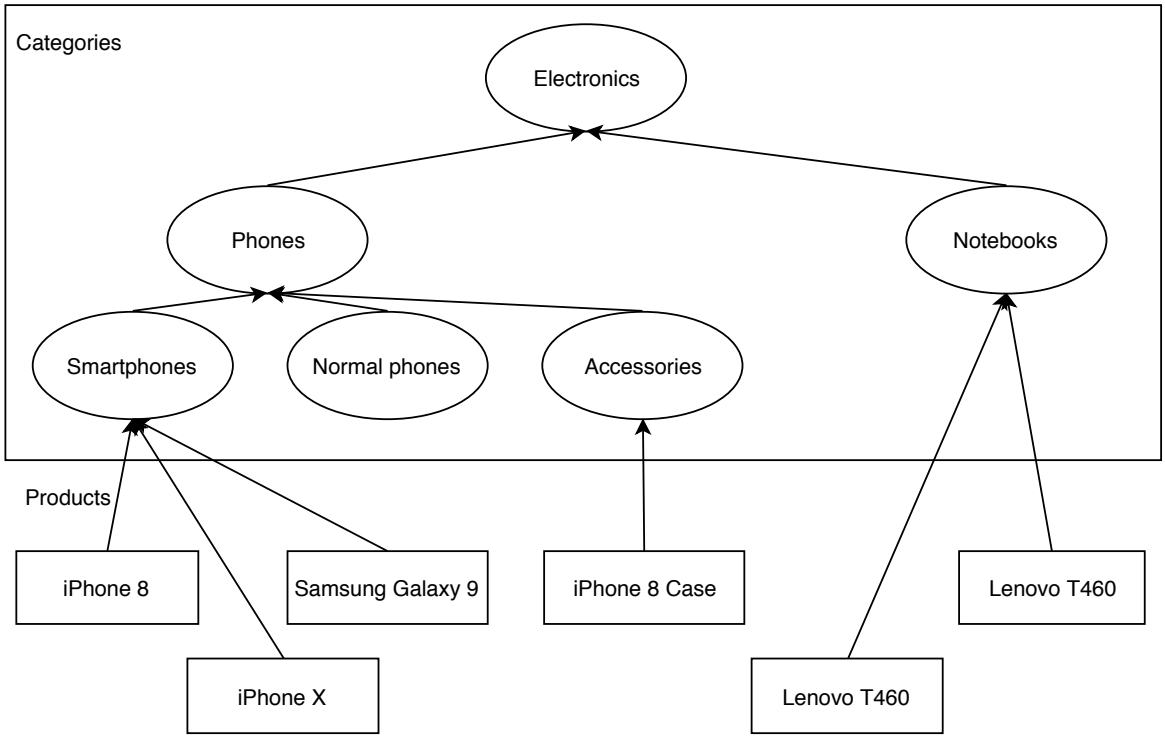



Figure 2.1: Basic category and product diagram

### 2.1.1 Data feeds

Data feeds from e-commerce systems are structured files which contain basic information about products and their attributes. In most cases they are used for ranking systems and contain only basic information: *name*, *manufacturer*, *category* and *price* for each product. Bellow is a short example from XML data feed.

```
<?xml version="1.0" encoding="UTF-8"?>
<ITEM>
```

```
   <NAME>Luxury set with curtains Mamo Tato</NAME>
   <ID>9082</ID>
   <ID_MAIN>9080</ID_MAIN>
   <CODE>11507903</CODE>
   <EAN>11507903</EAN>
      <URL>luxury-set-with-curtains</URL>
   <PRICES>
      <PRICE level="1">1586.40</PRICE>
   </PRICES>
   <PRICES_VAT>
      <PRICE_VAT level="1">1919.54</PRICE_VAT>
   </PRICES_VAT>
   <DELIVERY_TIME>in 7 - 10 business days</DELIVERY_TIME>
   <AVAILABLE>0</AVAILABLE>
   <DUTIES />
   <PRICE_BUY />
   <PRICE_OLD>0.00</PRICE_OLD>
   <VAT>21</VAT>
   <QUANTITY>0.000</QUANTITY>
   <LOCK>0.000</LOCK>
   <UNIT>pieces</UNIT>
   <WEIGHT>2200</WEIGHT>
   <IMAGES>
      <IMAGE description="Photo (18119)">https://www.shoply.cz/luxury-set-
      with-curtains18119.jpg</IMAGE>
      <IMAGE description="Photo (18120)">https://www.shoply.cz/luxury-set-
      with-curtains18120.jpg</IMAGE>
   </IMAGES>
   <CATEGORIES>
      <CATEGORY main="1" index="0005" id="352">Bed equipment</CATEGORY>
      <CATEGORY index="0005000300010001" id="514">Luxury sets</CATEGORY>
   </CATEGORIES>
   <CATEGORY_SHORT>Bed equipment | Sets | Luxury sets</CATEGORY_SHORT>
   <PARAMS>
      <PARAM name="Sheet size" type="3">140x70</PARAM>
   </PARAMS>
   <PRODUCER>Mamo Tato</PRODUCER>
   <SIGNS />
   <DESCRIPTION>Luxury set with curtains Mamo Tato</DESCRIPTION>
   <PRIORITY>5</PRIORITY>
   <SYNCHRONISM>1</SYNCHRONISM>
   <EXPORT>1</EXPORT>
   <ACTIVE>1</ACTIVE>
</ITEM>
```

#### 2.1.1.1 Existing feed formats

The biggest problem of data feeds is that there is no unified format. Some services like *Heureka.cz*[5] have a specific format which every system needs to implement. Creation of such data feed is always manual and the feed contains only limited amount of information. There are other formats used for affiliate programs like *Google RSS 2.0*[15] which contains also basic information, but the data in the format can be defined specifically to the system.

There has been a solution in the realm of feed unification, *ShopAPI.cz*[17] accumulates feeds from different vendors and offers feed conversion to one of the existing feed - such as the *Heureka.cz*[5] feed. But it is mostly used for ranking systems and affiliate programs.

### 2.1.2 E-commerce data

As we described in previous section products and categories are structured in the e-commerce environment. In figure 2.2 is a basic diagram of the database and how products and categories are usually linked together. From data feeds 2.1.1 and other e-commerce data we analyzed product attribute keys. Some product features are *common* (almost every product has this attribute key) and some are *specific* for given product or domain. In table 2.2 are listed all common product attribute keys and their raw type which were most frequent in analyzed dataset. We will use these attribute keys during the *scraping* of data described in section 2.5.

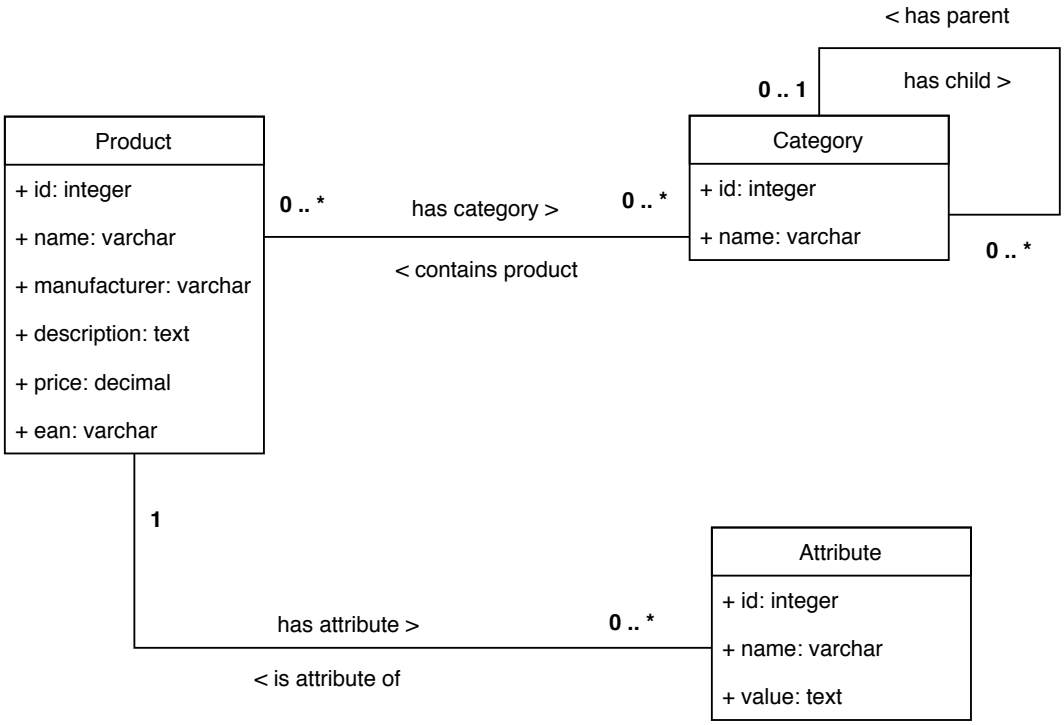



Figure 2.2: E-shop: Basic Entity Relation diagram

| Property key | Value type | Description |
|---|---|---|
| Name | string | name of the product |
| Category | string | product category |
| Price | double | price of the product |
| Manufacturer | string | product manufacturer |
| Description | string | product description |
| Short description | string | product short description |
| Currency | string | price currency |
| Weight | string | product weight with unit |
| Image | string | url of product image(s) |
| Color | string | product color(s) |
| Material | string | product material(s) |
| Depth | string | product dimensions - depth with unit |
| Width | string | product dimensions - width with unit |
| Height | string | product dimensions - height with unit |
| Size | string | product size(s) |

Table 2.2: Common Attribute keys table

#### 2.1.2.1 Product representation

Most important for searching or any other activities are the *specific* attributes. We can assume that products with similar or same specific property keys are also very similar to each other - probably are in the same category. Even then *category* can be considered as a property of a product. This view allows us to abstract product as a ***collection of properties*** (fig. 2.3) some of which are common with other products.

**Property weight** Even though a product is made from properties, not all have the same informational value. For basic querying, we can assume same weight among parameters because we are searching for specific results, but if we want to cluster the data into logical groups using these parameters not all of them can have the same weight.

## 2.2 Ontology and Semantic web

According to [25] ontology is specification of conceptualization. In other words, it is a description of things that exist and how they are related. Essentially when we talk about ontology it is a problem of knowledge management and data meaning. It is a way we can share meaning with each other, between user and computer and between computers themselves. It offers *modeling* and *interface* for different data structures. Ontology can be modeled by *OWL (Web Ontology Language)*.

### 2.2.1 What is Data?

In figure 2.4 we can see the *DIKW pyramid* of the data and knowledge. It describes a relationship between data, information, knowledge and wisdom. Information is defined from

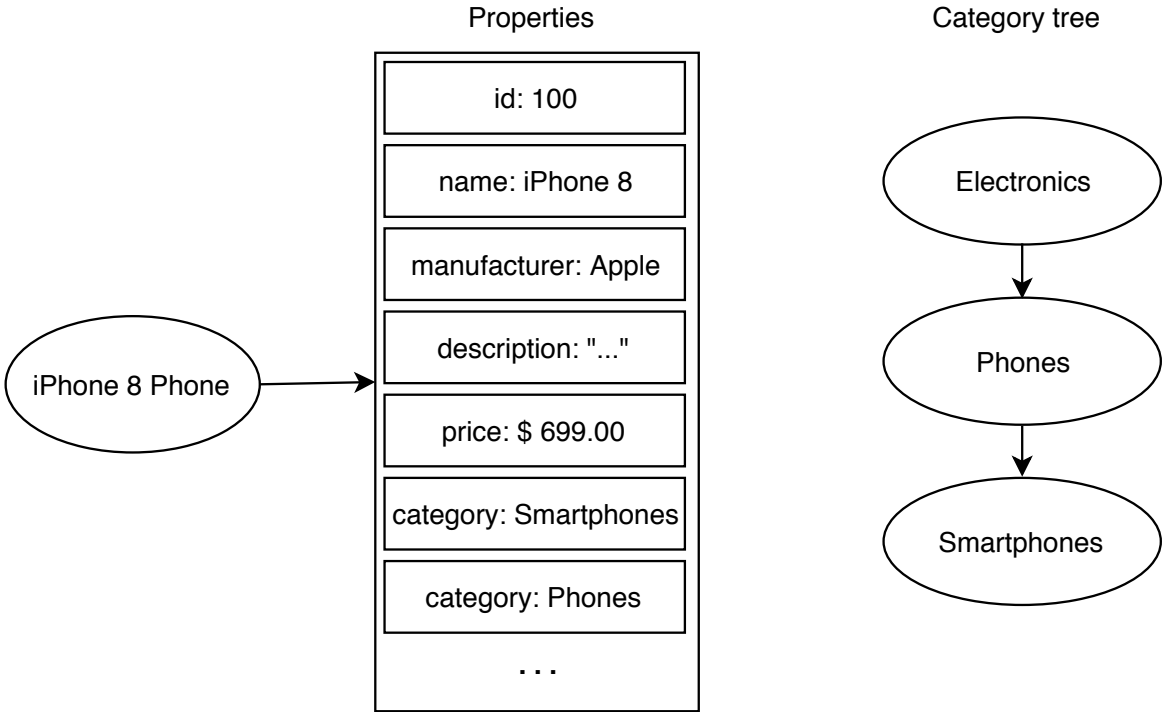



Figure 2.3: E-shop: Product represented as collection of properties

data and wisdom is defined from knowledge [28].

**Data as Fact or Symbol** [28] Data are discrete facts or observations without a context. Or can be viewed as symbols describing a property of an object, event or their environment.

**Information** In pyramid defined as knowledge by description. Information is data and meaning together.

**Knowledge** DIKW pyramid defines knowledge with reference to information - *processed information.*

**Wisdom** In paper [32] wisdom is described as *integrated knowledge* - information which is used usefully.

### 2.2.2 Semantic web

The semantic web is a traditional web, enhanced in a way that the data has specific meaning [30]. It is a concept which is used wildly by search engines and many websites like *Google* or *Amazon*. Many data types like dates and times are given the meaning of being a *Date* or *Time* which can help software and users to understand what the displayed

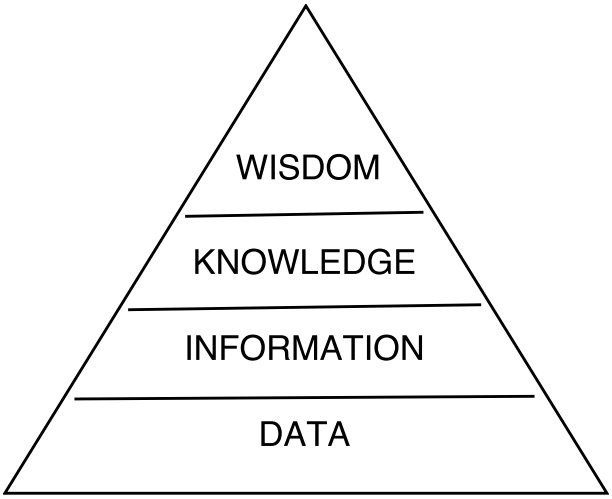



Figure 2.4: DIKW Pyramid - ROWLEY, Jennifer.
The wisdom hierarchy: representations of the DIKW hierarchy [29].

values mean. Ontology is an important part of the whole concept because it is a formal representation of the domain and the meaning.

Principles of semantic web are simple. Data are modeled by *RDF ((Resource Description Framework)* and *OWL*. Each resource has *IRI* - unique identifier for addressing web resources.

#### 2.2.2.1 RDF and RDF Schema

RDF is a meta-data model in form of *triple* **Subject-Predicate-Object** (figure 2.5). This triple then describes relationship between resources [13]. In figure 2.6 is displayed RDF representation. RDF store is an atomic decomposition of a graph between resources. RDF schema is a *vocabulary* which describes classes and properties with constrains used in RDF data.

Subject of the RDF is a resource defined by *IRI*[1] (Internationalized Resource Identifier) or a *blank node*[2]. The predicate is a relationship between subject and object and it is identified by URI which indicates resource representing a relationship. The object can be defined by resource IRI, blank node or string literal.

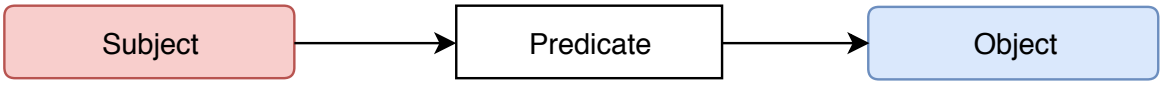



Figure 2.5: RDF - Resource Description Framework triple

[1] IRI is a generalization of URI (Uniform Resource Identifier)[19]
[2] Also called bnode or anonymous resource. It is a resource without IRI. Can be used only as subject or object.

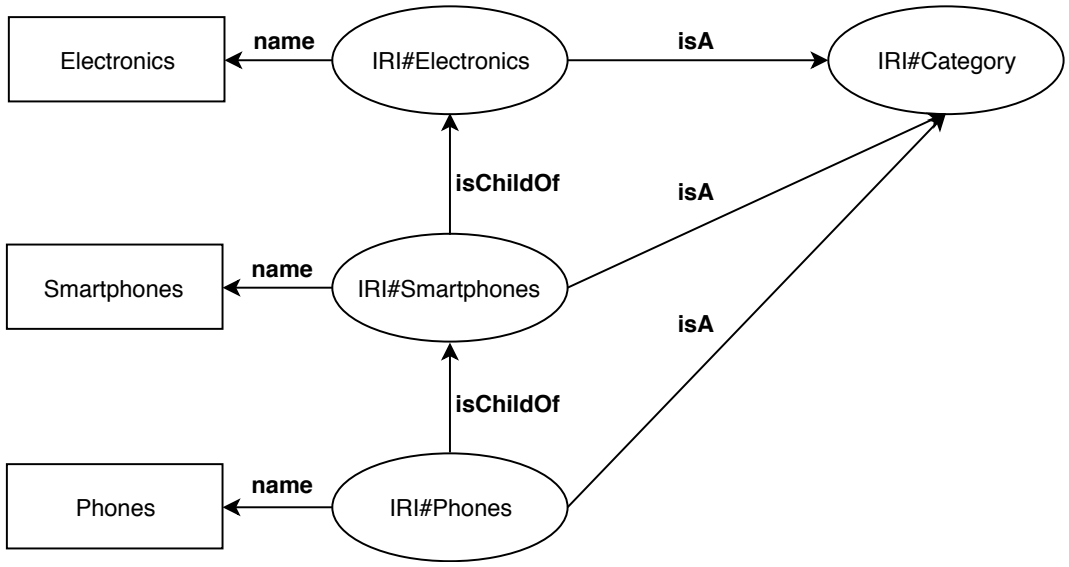



Figure 2.6: Example of data in RDF format

#### 2.2.2.2 OWL

Web ontology language is a markup language designed for ontology description. It expands properties of classes in RDF and RDF Schema [23]. OWL ontology is described using basic structures listed below. Ontologies are also defined by IRI.

- Class
- Object property
- Data property

Since OWL is extending RDF, classes, object properties and data properties are represented as resources and are defined by IRI. Instances of classes are called *individuals*. Each individual is defined by IRI has a predicate *rdf:type* connection to a class resource. OWL is making use of other properties such as *rdf:subClassOf* and *rdf:subPropertyOf* which enables hierarchical structures in the ontology [23]. Figure 2.7 shows RDF graph data defined using OWL.

**Classes** Every class is sub-class of *owl:Thing* and is a type of *rdf:Class*. Classes can be equal or disjoint with each other as well as form a hierarchical structure using the *subClassOf* object property. Every class can have additional description (annotations).

**Object properties** Each object property is sub-property of *owl:topObjectProperty* and of type *rdf:ObjectProperty*. Object properties can have characteristics which define more rules. These characteristics are summed in table 2.3. Individuals $A$ and $B$ being connected by

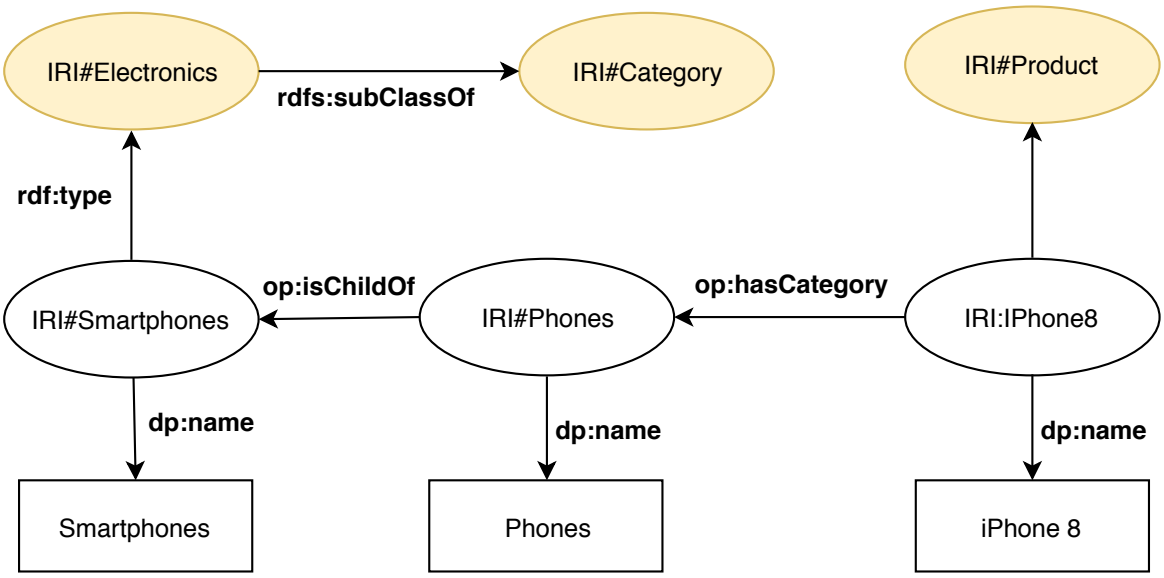



Figure 2.7: Example of data in RDF format

| Name | Description |
|---|---|
| Functional | For individual input there is only one output |
| Inverse functional | The individual output can be linked to only one input |
| Transitive | Transitive property - $A \rightarrow_p B \wedge B \rightarrow_p C \Longrightarrow A \rightarrow_p C$ |
| Symmetric | Property and it's inverse coincide |
| Asymmetric | If $A \rightarrow_p B$ then $B \rightarrow_p A$ cannot exist |
| Reflexive | Property relates everything to itself |
| Irreflexive | No individual can be related to itself |

Table 2.3: Object property characteristics

property is denoted as $A \rightarrow_p B$. Object property can also be an *inverse* of another property. For example property *hasParent* has inverse *hasChild* [23].

Object properties also can have *domain* and *range* definition. Domain and range are defined from the set of classes - it can be individual class, class intersection or union.

**Data properties** Every data property is sub-property of *owl:topDataProperty* and of type *rdf:DatatypeProperty*. Same as object properties *domain* and *range* can be defined. Since data properties are literal value, domain is from set of classes and range can be a string value of defined type such as *xsd:string* or *xsd:double* [23].

**Serialization format** OWL ontology with individuals can be serialized into multiple formats. Most used formats are *RDF/XML*, *Turtle* or *JSON-LD*. Listing 2.1 shows example of OWL class, data and object property definition in turtle format.

```
### Object property example
```

```
###  https://dev.novotmike.com/oes#hasAttribute
:hasAttribute rdf:type owl:ObjectProperty ;
              owl:inverseOf :isAttributeOf ;
              rdfs:domain :Product ;
              rdfs:range :Attribute .

### Data property example
###  https://dev.novotmike.com/oes#name
:name rdf:type owl:DatatypeProperty ;
      rdfs:domain [ rdf:type owl:Class ;
                    owl:unionOf ( :Category
                                  :Product
                                )
                  ] ;
      rdfs:range xsd:string .

### Classes example
###  https://dev.novotmike.com/oes#Attribute
:Attribute rdf:type owl:Class .

###  https://dev.novotmike.com/oes#Product
:Product rdf:type owl:Class .
```

Listing 2.1: OWL structure examples in Turtle format

#### 2.2.2.3 OWL: Resoning

Individuals and ontology definition is stored in one RDF graph. This enables *reasoning* over the data. Reasoner is a program which can infer new properties to an individual based on the OWL defined or user-defined rules. Reasoner can also check *consistency* of the ontology, meaning that when we add new individual it's properties fit the defined ontology. In semantic data we recognize two concepts which answer to missing data meaning [30]:

**OWA: Open World Assumption** Everything that cannot be proven is unknown.

**CWA: Close World Assumption** Everything that cannot be proven is false.

The OWA and CWA concepts can tell the system how to react when something is unknown. *OWA* answer is undefined - which means *Not known*. The *CWA* answer is *no* - answers that the assumption is false.

**SWRL - Semantic Web Rule Language** SWRL is a language which enables defining of additional rules. Example of such rule can be a *hasCategory* property of a *product*. When product belongs to a category which is a *child of* another category, then it belongs to the parent category as well. This can be expressed by the rule 2.1. Using this rule when product

$p$ has category individual $c1$ and that individual is child of some other category $c2$ then product $p$ has a category $c2$. Application of this rule can be seen in figure 2.8.

$$hasCategory(?p, ?c1) \wedge isChildOf(?c1, ?c2) \rightarrow hasCategory(?p, ?c2) \tag{2.1}$$

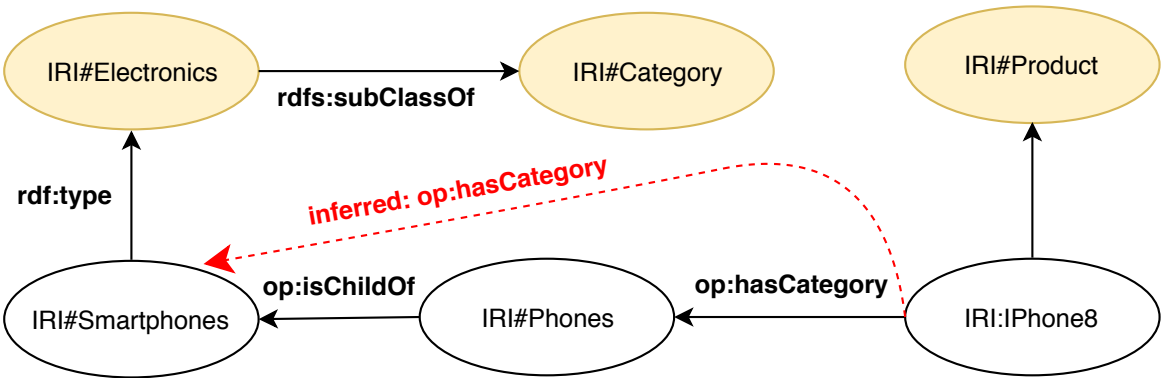



Figure 2.8: OWL reasoner inferred object property diagram

SWRL rules can be more complex than example 2.1, but do not allow for negations. Furthermore $OR$ relationship is represented by two or more rules based on the number of branches. Rules can also imply the class defined in owl, in this case, rule 2.1 would be enhanced to class representation 2.2

$$\begin{aligned} &Product(?p) \wedge Category(?c1) \wedge Category(?c2) \wedge \\ &hasCategory(?p, ?c1) \wedge isChildOf(?c1, ?c2) \rightarrow hasCategory(?p, ?c2) \end{aligned} \tag{2.2}$$

**Reasoners** There are multiple implementations of semantic reasoner many of which are free and open source. Table 2.4 lists some of the reasoners which were considered during the analysis. All reasoners are compatible with OWL API [9] which is a Java API for building, managing and serializing OWL ontologies.

| Name | Description | License |
|---|---|---|
| FaCT++ reasoner[3] | OWL and OWL 2, Consistency checking | LGPL license |
| Pellet[10] | OWL 2, Java based, SPARQL, Consistency checking | AGPL 3.0 license |
| Hermit[4] | OWL and OWL 2, Consistency checking | LGPL license |

Table 2.4: Semantic reasoners

### 2.2.3 Defining ontology

Ontology can be defined from scratch or by modifying existing ontology. A process for modeling an ontology is called *ontology engineering* [26]. As we described in previous section 2.2.2, ontology is defined using OWL and to define new ontology we need to define classes,

object properties, data properties and their hierarchy. We followed the process described in [26]. Steps are described in following paragraphs.

**1. Ontology domain and scope** Ontology domain and scope need to be defined first. We can use existing ontologies from similar or related domains. The scope is important because it defines the granularity of the classes. Next, we enumerate all important terms in the domain - from this set of terms, we will create classes and properties.

**2. Class and hierarchy definition** We apply *top-down*, *bottom-up* or combination of both approaches to get all classes and their hierarchy. Top-down approach starts from most general terms, while bottom-up approach starts from most specific.

**3. Class properties** Classes are described by the properties. We need to find all the properties from the terms and decide which classes are defined by them. In addition, we need to assign the range of the property (a property value domain). We also need to think about the cardinality of the properties. If a class should have more than one property of one type, maybe we should abstract it as a class.

**4. Class relationship definition** In the last step, we need to define how classes relate to each other. Some relationships come up from class properties other from the definition of the domain.

#### 2.2.3.1 Modeling tools

There are lots of tools for ontology modeling. For purposes of this project, we will use *Protégé* [12]. Protégé website offers documentation and manuals on how to build ontologies. Main functions of ontology modeling tools are summed in the following list.

- Tools for ontology modeling
- Individual creation
- Reasoning over ontology
- Consistency checking using a reasoner
- SPARQL support for querying
- Ontology serialization and deserialization

## 2.3 Methods for product similarity and relation

In this section, we will describe several methods which can be used to infer product similarity and enable product grouping based on the properties of the products.

### 2.3.1 Product similarity methods

Product similarity can be inferred in different ways. We have stated that product can be represented as a set of attribute key-value pairs. We are going to analyze methods which would exploit this representation for similarity factor calculation.

We can represent a product as a set of only attribute keys and create a *product vector* with dimension equal to the size of the collection of all analyzed attribute keys. A product would be then represented by a vector which for each attribute key $i$ contains value based on equation 2.3.

$$v(i) = \begin{cases} 1 & \text{if product contains key} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

**Euclidean distance** Simplest algorithm for similarity calculation is Euclidean distance. Distance of two products $p$ and $q$ using our product vector representation $p_v$ and $q_v$ is calculated using Euclidean distance equation 2.4. We assume that $n$ is the size of the set of all attribute keys.

$$d(p_v, q_v) = \sqrt{\sum_{i=0}^{n} (q_v(i) - p_v(i))^2} \tag{2.4}$$

**Weighed euclidean distance** We can enhance simple Euclidean distance calculation by introducing weights for each attribute key. The product is represented in the same way as in classic euclidean distance algorithm. We only add new vector of weights $w(i)$ based on the attribute key informational value (as described in section 2.1.2.1). Equation 2.5 shows weighed Euclidean distance calculation of two products $p$ and $q$.

It depends on the way weights are assigned for each attribute key. We will discuss this further in section 3.2.2.1.

$$d(p_v, q_v) = \sqrt{\sum_{i=0}^{n} w_i (q_v(i) - p_v(i))^2} \tag{2.5}$$

**Cosine similarity** Cosine similarity is another method for distance calculation of two non-zero vectors. It is independent of vector length [33]. Similarity between two vectors $p$ and $q$ is calculated using equation 2.6. Resulting value is in interval: $\langle -1, 1 \rangle$.

$$S_{p,q} = cos\alpha = \frac{p_v^T q_v}{||p_v|| \cdot ||p_v||} \tag{2.6}$$

**Property based similarity** We can also calculate specific attribute value similarity. For string values such as *categories* and *manufacturers* we can use exact matching or *Levenshtein distance* metric to express the similarity of two values. For numeric values we can use equality or interval matching for similarity calculation. Additionally if we have information about the value unit (weight or dimensions) we can convert units if necessary.

**Levenshtein distance** Levenshtein distance between two strings $a$ and $b$ is calculated using equation 2.7. Function $1_{(a_i \neq b_j)}$ is 0 when $a_i = b_j$ and 1 otherwise. The resulting distance represents number character operations (insert, delete or substitute) required to change one string to another.

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i1,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise} \end{cases} \tag{2.7}$$

### 2.3.2 Product clustering methods

Clustering is a technique of data grouping based on the nature of the data properties. Generally, a clustering method is given data points (data represented as points in space) and each point is assigned to specific group [33]. Each group represents a cluster of similar data.

**K-Means clustering** K-means clustering is a method which takes a set of data points as input and groups the data points into $k$ clusters based on the nearest *mean* which represents a prototype of the cluster. The number of clusters is limited by the number of data points ($k \leq n$). Algorithm steps are described in the following list and in diagram 2.9.

1. Define $k$ classes and create their representation in space. These points are called center points.
2. Each data point is classified by computing distance from each center point. The Data point is assigned to a group which is represented by the nearest center point.
3. Recompute center points for each group using the *mean* of all vectors assigned to the group.
4. Repeat steps 1-3 until the center points position does not change or change is minimal.

**Advantages**
+ Simplicity
+ Complexity [33] $O(nkdi)$[1]

**Disadvantages**
- Upfront class number definition
- Different results depending on the initial center points

1 $n$ vectors of dimension $d$, $k$ clusters and $i$ iterations

**Hierarchical clustering** Hierarchical clustering can be divided into two categories: *top-down* or *bottom-up*. Bottom-up algorithms treat each *data point* as individual cluster. Cluster pairs are merged until all data is in one big cluster. Cluster hierarchy is represented as a tree. Following list describes algorithm steps and diagram 2.10.

1. Each point is represented as a cluster

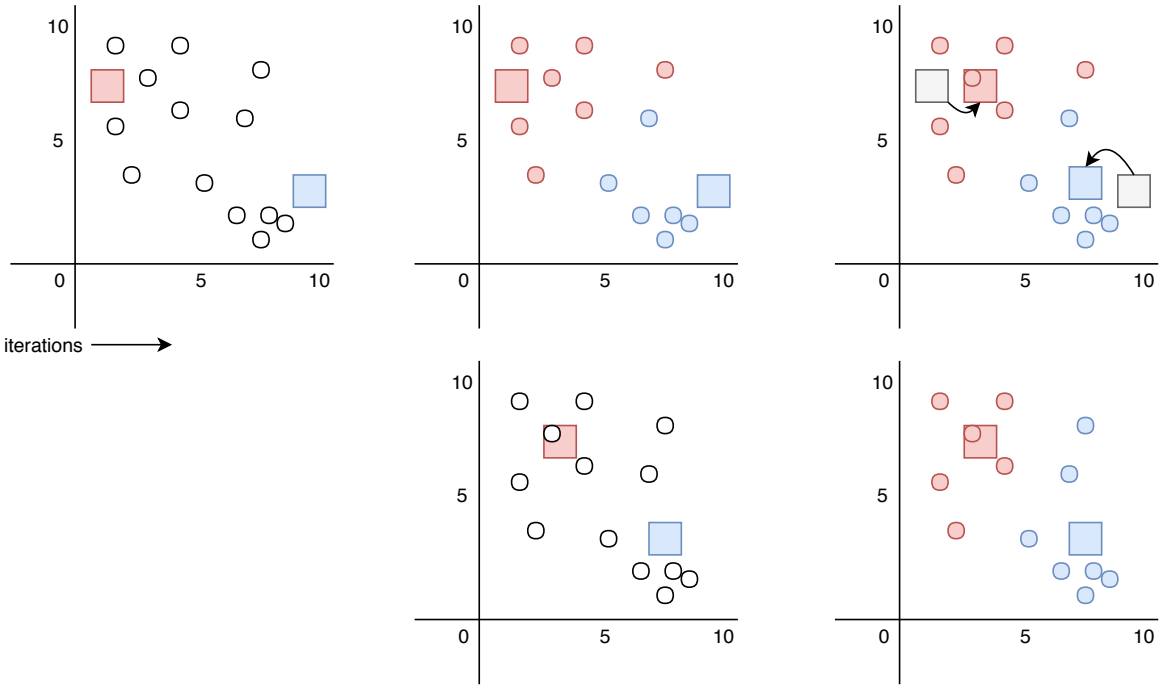



Figure 2.9: K-Means algorithm diagram

2. We combine two clusters into one depending on the distance between clusters. This distance is calculated as *average* distance between points in the first cluster and the second cluster. In each step algorithm chooses clusters to merge based on the minimal distance.

3. Repeat step *2* until data is clustered into one cluster.

**Advantages**
+ No upfront class number definition
+ Not sensitive to distance metric selection
+ Good for data hierarchy re-building

**Disadvantages**
- Time complexity $O(n^3)$

## 2.4 Data storage

E-commerce data such as product, category and order information is usually stored in a structure with a well-defined *schema*. Other information such as user activity logs, images and other can be often without structure. Every type of data needs to be stored in a way to be simply retrieved in the system - usually using some database system. Storage can be done in one of two types of database systems: *SQL* databases or *NoSQL* databases.

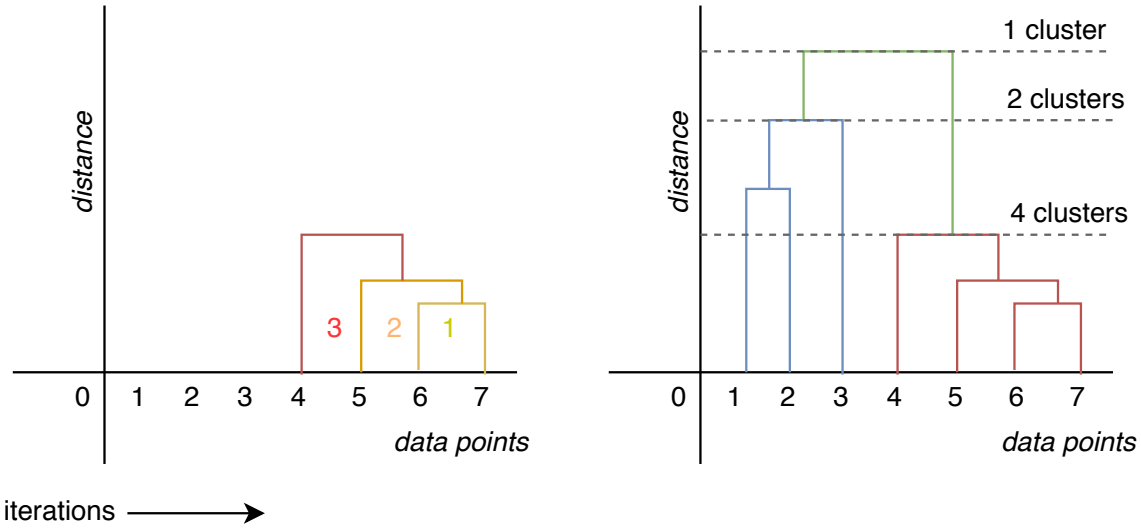



Figure 2.10: Hierarchical agglomerative clustering diagram

### 2.4.1 SQL Databases

SQL (Structured Query Language) databases are a classic option for storing data. They offer solid schema and structure for data and have been tested by the time. E-commerce systems tend to use SQL databases for critical data - transactions, orders and other. This is because of the nature of so so-called *ACID* - (Atomicity, Consistency, Isolation, Durability) [31]. This means that the can be stored and manipulated safely and with exact results.

### 2.4.2 NoSQL Databases

NoSQL databases are a more modern approach to data storage. There are many NoSQL systems spanning from simple *key-value* databases such as *Redis* [14] which are suitable for storing things like shopping carts or temporary data, all the way to graph databases like Neo4J [7]. NoSQL databases are generally used to store unstructured data since they usually do no support or require schema definition. Their biggest advantage is horizontal scalability.

**CAP Theorem** CAP theorem mentions that there is a trade-off between *Consistency*, *Availability* and *Partition tolerance*. NoSQL system can only really guarantee two of these aspects [31].

- *Consistency* - data is consistent on every node of replication
- *Availability* - Data must always be accessible
- *Partition tolerance* - data is accessible despite the partitioning or communication failure

Database can be *CA*, *CP* or *AP* depending on which aspects of CAP theorem they guarantee.

### 2.4.3 SQL vs NoSQL Databases

In reality, most e-commerce solutions use a combination of SQL and NoSQL databases. Advantages and disadvantages are summed in table 2.5. General SQL and NoSQL system advantages and disadvantages are noted in paragraphs below. SQL systems are reliable, stable and allow for complex querying. Disadvantages of SQL are bad scalability with multiple access points. The advantage of NoSQL is the ability to scale, to store unstructured data and speed. A disadvantage is that NoSQL databases are still in development and do not offer the kind of stability SQL system does.

### 2.4.4 Storing and retrieving semantic data

There are different options for storing semantic data. The usual way of storing ontologies is a RDF triple store, but these stores are mainly for exchanging data and are not really designed for efficient data storing [21]. But there is a relation between RDF stores and general graph databases which represent labeled property graphs (LPG). We can also consider classic SQL database such as *PostgreSQL* [11]. We have chosen two representatives - PostgreSQL and Neo4J and compared them with regards to semantic data storing and our scenarios 1.2. Advantages and disadvantages of each database type representative are summed in table 2.5.

**Querying** As far as querying of the data is concerned RDF stores, Neo4J and PostgreSQL all have powerful querying languages. RDF stores have SPARQL [23], Neo4J has cypher [7] and PostgreSQL has classic SQL. All enable complex querying. Neo4J, in addition, offers *traversal framework* which enables path directed querying (specified movement in a graph).

**Comparison** SQL system does not allow for easy extensibility which is a nice feature of ontologies in general. Because of that graph database is more suitable for storing semantic data. We will talk about how to store RDF semantic data to a graph database in design section 3.2.2.3.

| Database system | Type | Advantages | Disadvantages |
|---|---|---|---|
| PostgreSQL[11] | SQL | ACID, SQL querying, well defined schema, transaction | Not easily extensible, only vertically scalable |
| Neo4J[7] | NoSQL | CA, CYPHER querying, transactions, simply extensible, horizontally scalable | no schema, data stored in one graph |

Table 2.5: Database systems and their advantages and disadvantages for storing semantic data

## 2.5 Data scraping

Data feeds are not only way how to retrieve data from an e-commerce website. On one hand, if you have an existing database with data, you can use basic queries to retrieve

information and store them in the data feed. On the other hand, if you need to get data from online shop, for example for learning purposes, you need some *data scraping tool*. Data scraper, in this case, is usually a simple tool that crawls the website and stores the relevant information. There are multiple tools like *Scrapy*[16] or *Apify*[1] which enable user to crawl a website and retrieve the needed data.

A disadvantage of just using the tools is that they are not built for every website and you have to specify a script which will retrieve the data. The best way would be to design and create own crawler which is also able to use machine learning to find relevant information and retrieve them.

### 2.5.1 Scraped data feeds

For purposes of this thesis we scraped data from on-line shops *Alza.cz* and *Kasa.cz*. We focused on specific category of products: *notebooks*, *phones* and their *accessories*. We were able to collect over *19000* products from both websites. Steps of the HTML scraping are described in the following list.

1. Listing all products in a category on the web page (or go page, by page)
2. Getting product detail URLs and storing them in a file
3. Using *wget* command to download all the HTML files using URLs acquired in the previous step

Data scraping was implemented using *Beautiful Soup* python library and python language. We processed each data domain separately and stored the data in the *unified feed format* (see section 3.1.2) to a *YAML* file.

# Chapter 3

# Design

As we described in sections 1.1 and 1.2 we need to design a system which will take data from different streams (data feeds), transfers it to unified format and then filter and clean the data in order to store them in a structure which will provide good querying capabilities. In following sections, we will discuss the whole process of data transformation, knowledge acquisition and storage. We also describe the architecture of the system.

## 3.1 Data structure

After analyzing our data 2.1 we defined basic *common* attributes (listed in table 2.2). We built our ontology based on these attributes.

### 3.1.1 Ontology

We used process defined in 2.2.3 to build the ontology. There were many iterations of the ontology from more specific to more general. The final version of the ontology describes products, categories, attributes and relations between them in a non-domain specific way. Tables 3.1, 3.2 and 3.3 list all classes, object properties and data properties defined in the ontology.

The ontology was designed based on the data which were available for this thesis. It may not contain all available classes in e-commerce domain, but it can be easily expanded in the future adding more classes, properties or using existing ontologies for more information.

| Class name | Description |
|---|---|
| Product | a product |
| Category | product category |
| Attribute | general attribute |
| Manufacturer | a product manufacturer |
| Price | price of the product |
| Currency | currency unit of the price |
| Unit | unit of a specific attribute |
| Dimensions | product dimensions (width, height, depth) |
| DimensionUnit | dimensions unit |
| Weight | product weight |
| WeightUnit | a unit of weight |

Table 3.1: Ontology classes

| Object property | Inverse | Domain | Range |
|---|---|---|---|
| hasAttribute | isAttributeOf | Product | Attribute |
| isInCategory | isCategoryOf | Product | Category |
| hasChild | isChildOf | Category | Category |
| hasCurrency | isCurrencyOf | Price | Currency |
| hasDimensions | isDimensionOf | Product | Dimensions |
| hasManufacturer | isManufacturerOf | Product | Manufacturer |
| hasPrice | isPriceOf | Product | Price |
| hasWeight | isWeightOf | Product | Weight |
| hasUnit | — | Weight,Dimensions, Attribue | WeightUnit, DimensionUnit, Unit |
| isCompatibleWith | — | Product | Product |
| isSimilarTo | — | Product | Product |

Table 3.2: Ontology object properties

| Data property | Domain | Range |
|---|---|---|
| name | Category, Currency, Product, Manufacturer, DimensionUnit, WeightUnit | xsd:string |
| key | Attribute | xsd:string |
| val | Attribute, Weight | xsd:string, xsd:double |
| width | Dimension | xsd:string, xsd:double |
| height | Dimension | xsd:string, xsd:double |
| depth | Dimension | xsd:string, xsd:double |
| amount | Price | xsd:double |
| short_description | Product | xsd:string |
| description | Product | xsd:string |
| sign | Currency | xsd:string |
| code | Currency | xsd:string |

Table 3.3: Ontology data properties

### 3.1.2 Unified format

With regards to our ontology described in the previous section, we need to prepare our data in a way that will enable simple manipulation and knowledge extraction. In section 2.1.2 we described how data is structured usually in the e-commerce system. Then in 2.1.2.1 we talked about a product as a collection of properties. We propose a *unified feed structure* which represents products simply as a collection of string key-value pairs. We chose *YAML* (YAML Ain't Markup Language [20]) as our serialization format because of it's better human-readability. Listing 3.1 shows an example of data in the unified format.

```
products:
- attributes:
  - key: "name"
    value: "Alfred Dunner Essential Pull On Capri Pant"
  - key: "description"
    value: "You'll return to our Alfred Dunner pull-on capris again
    and again when you want an updated, casual look and all the
    comfort you love. elastic waistband approx. 19-21.
        inseam slash pockets polyester washable imported"
  - key: "price"
    value: "41.09"
  - key: "currency"
    value: "$"
  - key: "category"
    value: "alfred dunner"
  - key: "categorytree"
    value: "jcpenney|women|alfred dunner"
  - key: "rating"
    value: "4.7 out of 5"
  - key: "image"
    value: "http://s7d9.scene7.com/is/image/JCPenney/DP1228201517142050M
    .tif?hei=380&amp;wid=380&op_usm=.4,.8,0,0&resmode=sharp2&op_usm=1.5
    ,.8,0,0&resmode=sharp"
  - key: "brand"
    value: "Alfred Dunner"
```

Listing 3.1: Unified feed YAML example

## 3.2 Solution design

With ontology defined in section 3.1.1 and data in unified format we propose a system consisting of two main components: *Feed component* and *Consolidation component*. Both components create a library which takes in data (3.1.2), performs knowledge extraction and stores semantic data into a database.

### 3.2.1 Feed component

Feed component of the library takes care of the initial data manipulation, filtering, and annotation for knowledge extraction. Since none of our data (listed in table 2.1 originally comes in the unified format first step of the process is conversion. Each step of the process is described in following sections. Top level view of the feed component is shown in figure 3.1.

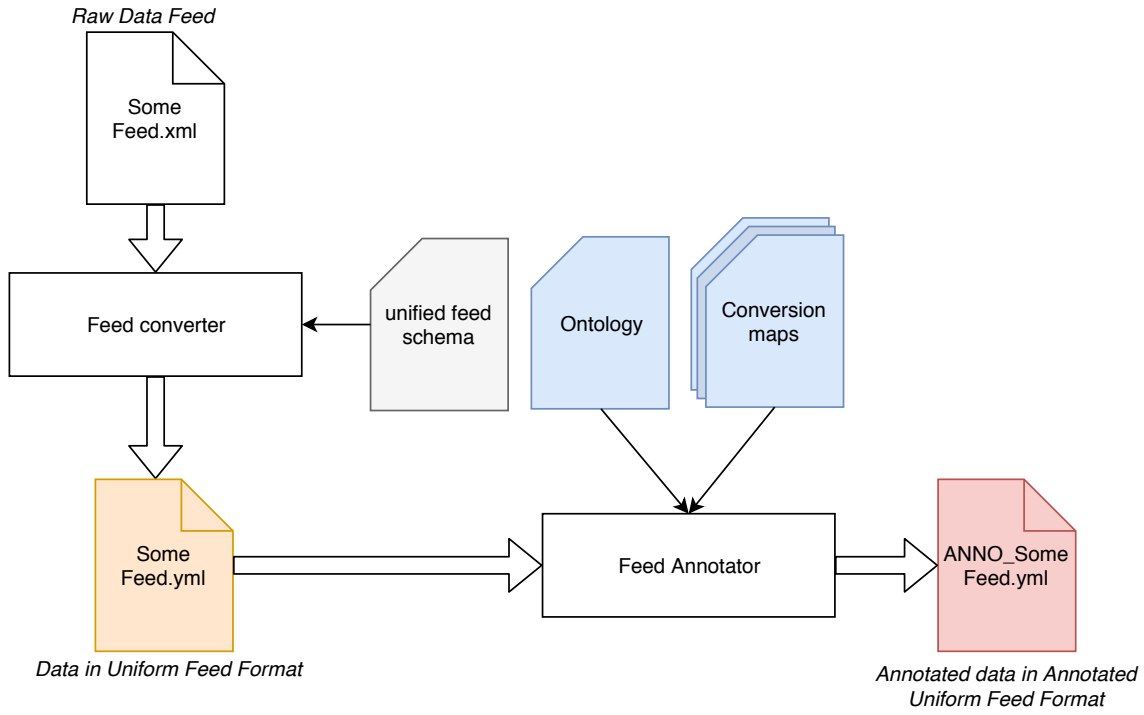



Figure 3.1: Top level diagram of feed component

#### 3.2.1.1 Converting data

Conversion of existing data feeds is a simple process. Because the files have a clear structure it is easy to design a converter which takes feed as an input and outputs the same data in the unified format.

As for the scraped data from 2.5 we need to design specific programs which will extract the data from product detail HTML pages. Most e-shops have a *name*, *description* and *price* in specific places. In addition they contain a table with all additional product properties.

#### 3.2.1.2 Conversion maps and vocabularies

After the data conversion, we analyzed the attribute keys and values. We were able to match keys which represent equal attribute type. This allowed us to create a *conversion map* which can assign attribute type class based on the string value of the key.

We also defined conversion maps and vocabulary of known types for *unit matching*, *currency matching* and *unit value conversion*. These maps are stored in separate files and can be easily changed or extended.

#### 3.2.1.3 Data annotation

Using the conversion maps we can defined multiple functions which take in attribute key-value pair and return annotated key-value pair. We annotated common product attributes listed in 2.2. Additionally we defined functions which extract *units* from *dimension* and *weight* values and *currency* from *price*. Tables 3.4 and 3.5 list all units and currencies which are considered during annotation process. Diagram 3.2 shows ho attribute key-value pair is processed.

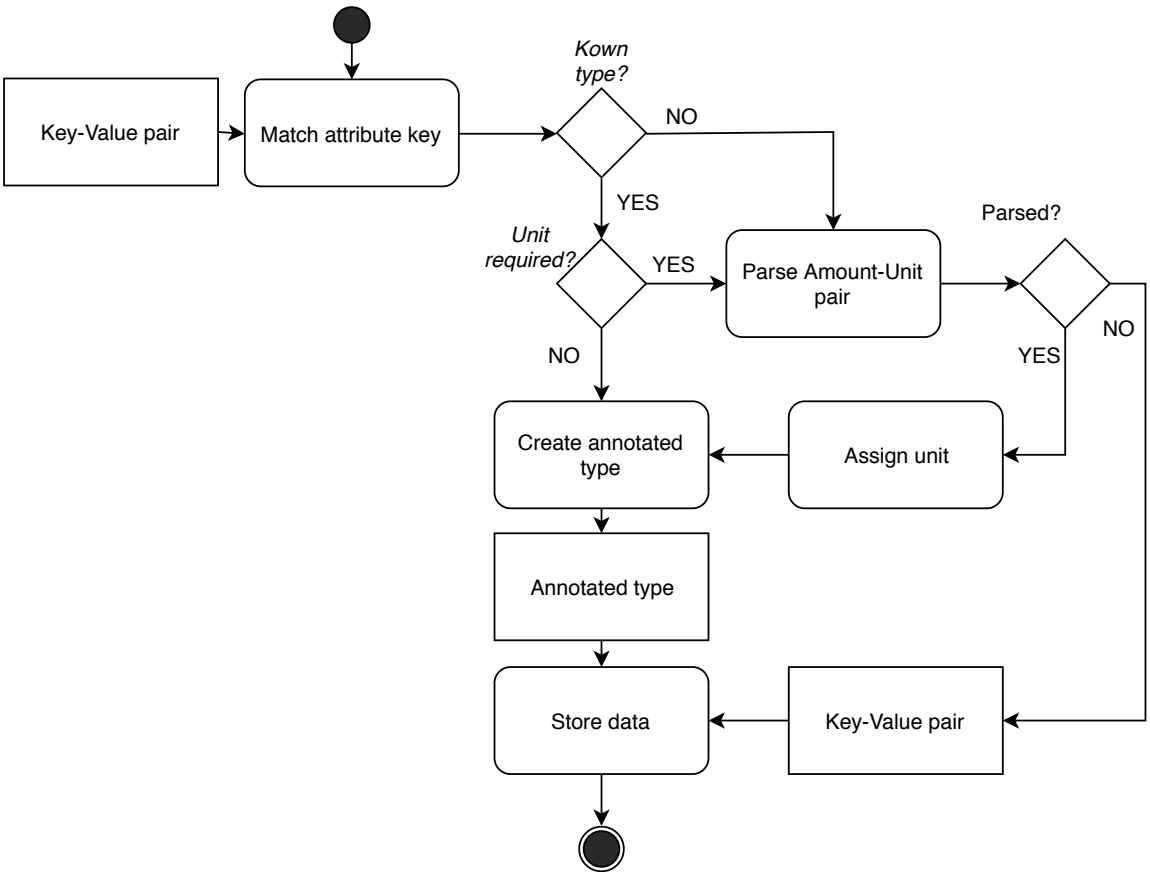



Figure 3.2: Attribute annotation processing diagram

**Annotation process** For each attribute in a product in the unified format a *key-value* pair is taken. The *key* is evaluated and annotated with corresponding *class* (classes are described in table 2.2) according to conversion function. Some of the classes such as *manufacturer* or *category* are just annotated, while other attributes such as *price*, *weight* or *dimensions*

| Description | Annotation name |
|---|---|
| Czech crown, czk | CCZK |
| US dollar, usd | CUSD |
| British pound, gbp | CGBP |
| Euro, eur | CEUR |

Table 3.4: List of currency types and their annotation

| Description | Unit type | Annotation name |
|---|---|---|
| Tonne, t | Weight | UWT |
| Kilogram, kg | Weight | UWKG |
| Gram, g | Weight | UWGR |
| Milligram, mg | Weight | UWMG |
| Kilometer, km | Dimension | UDKM |
| Meter, m | Dimension | UDMTR |
| Decimeter, dm | Dimension | UDDMT |
| Centimeter, cm | Dimension | UDCMT |
| Millimeter, mm | Dimension | UDMMT |
| Inches, ¨ | Dimension | UDINCH |
| Gigabytes | Digital information | UCSGB |
| Megabytes | Digital information | UCSMB |
| Kilobytes | Digital information | UCSKB |

Table 3.5: List of units, their annotation and domain

are also tested for their value and value format. These attribute are based on the ontology defined in 3.1.1.

**Attribute units** Some of the most interesting attributes have a value containing numeric *amount* and a *unit*. We use dedicated programs to parse the amount and unit from the attribute value. These programs use currency and unit vocabularies to match known units and annotate them accordingly.

The following list shows common attributes which should contain amount-unit pair in their value.

- Price - contains currency unit
- Weight - contains unit of weight
- Dimensions: width, height, depth - contain unit of dimension

The dedicated programs will only annotate the attribute if the unit and amount are parsed correctly and the unit exists in the vocabulary of known units.

**Unit conversion** All units of the common attributes during the annotation process should be unified in order for the data to be consistent. Because we annotate the data with units we can easily convert the values using a *unit conversion map*. Table 3.6 lists base unit for a given unit type.

| Unit type | Base unit |
|---|---|
| Currency | US Dollar |
| Weight | Grams |
| Dimension | Centimeters |
| Digital information | Megabytes |

Table 3.6: Unit types with basic unit for annotation

**Price matching** In order for the price to be matched data needs to contain currency attribute or currency must be specified in price attribute value. Currency is parsed from price value using a program which uses *currency conversion map* and vocabulary in order to annotate the value. Processing diagram 3.3 shows input and output of the matching program.

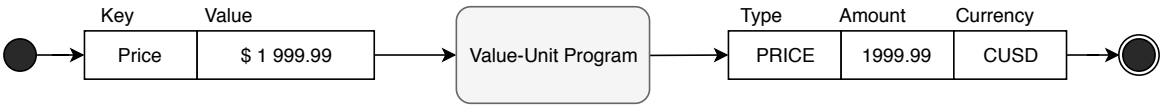



Figure 3.3: Price attribute processing diagram

**Weight and Dimensions matching** Weight and dimensions attributes are matched in a similar way the price is using a dedicated program. Dimensions - *width*, *height* and *depth* are clustered together and all have to have the same unit. If the units do not match, they are converted using *unit conversion map*.

**Specific attributes** Rest of the domain or individual specific attributes are either matched using our programs as *amount-unit* pair or are passed along without annotation.

**Annotated unified format feed** Annotated data is stored in annotated unified format feed. A product is represented as a set of annotated common attributes with matched units, set of specific attributes with matched units and set of unmatched specific attributes. This feed is a base for consolidation component or further filtering or can be used for additional attribute matching (new attributes or domain specific attributes). Listing 3.2 shows an example of annotated feed created from the feed shown in listing 3.1.

```
products:
- attributes:
  - key: "categorytree"
```

```
    value: "jcpenney|women|alfred dunner"
  - key: "rating"
    value: "4.7 out of 5"
  name:
    name: "Alfred Dunner Essential Pull On Capri Pant"
  categories:
  - name: "alfred dunner"
  manufacturer:
    name: "Alfred Dunner"
  price:
    price: 41.09
    currency: "CUSD"
  description:
    shortDescription: null
    description: "You'll return to our Alfred Dunner pull-on capris again
    and again when you want an updated, casual look and all the
    comfort you love. elastic waistband approx. 19-21. inseam slash pockets
    polyester washable imported"
  images:
  - url: "http://s7d9.scene7.com/is/image/JCPenney/DP1228201517142050M
  .tif?hei=380&amp;wid=380&op_usm=.4,.8,0,0&resmode=sharp2&op_usm=1.5
  ,.8,0,0&resmode=sharp"
    title: null
  dimensions: null
  weight: null
  materials: null
  colors: null
  sizes: null
```

Listing 3.2: Annotated unified feed YAML example

#### 3.2.1.4 Vocabularies

We used annotated data to create *vocabularies* for *manufacturers*, *categories*, *units* and *currencies*. These vocabularies contain unique values and will be used during the consolidation process in order to create individuals for each representative. This will allow us to interconnect product witch same *manufacturers* or *categories*.

### 3.2.2 Consolidating component

Consolidation component of the *Feeder library* loads in product data in *annotated unified data format*. The main purpose of this component is to use the ontology and build individuals in the ontology based on the data feed. It also performs multiple operations such as product relation building and final storage of the data. Top-level diagram 3.4 shows how consolidation component is designed.

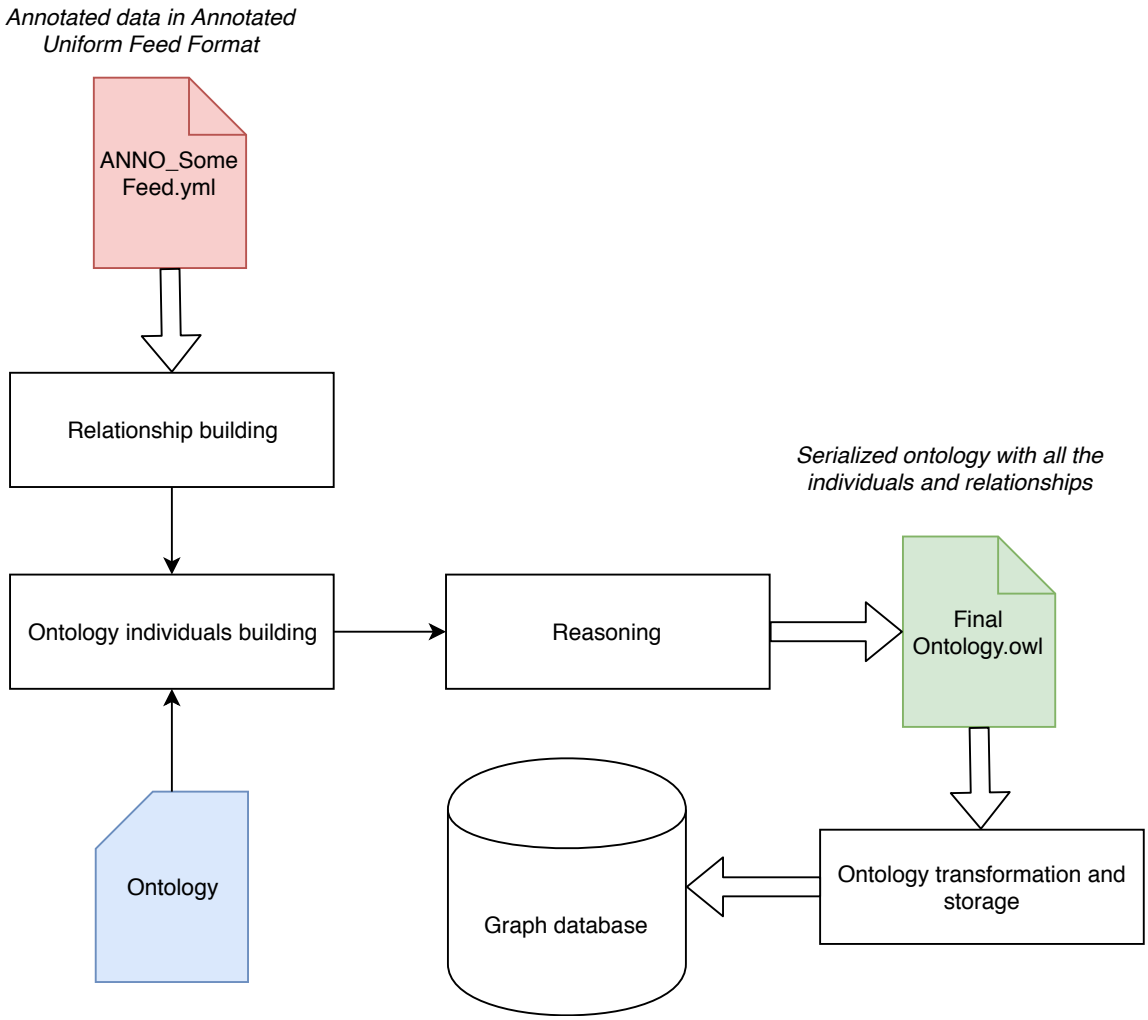



Figure 3.4: Top level diagram of consolidation component

#### 3.2.2.1 Product relation building

In section 2.3 we discussed few possible algorithms for product relationship building. We are considering relationships *isSimilarTo* and *isCompatibleWith* which are based on the ontology 3.1.1. We describe few possible product vector representation which enable creation of relationships based on the similarity algorithms.

**General similarity** A vector representation of the product described in section 2.3 can be used to calculate the general similarity between products. We use *euclidean distance* algorithm and a threshold value $t$. If the distance is smaller than the threshold value $d(p_v, q_v) < t$ the product are considered generally similar. Vector is created from specific attribute keys only - we create a set of all specific attributes and the resulting vector contains values based on the presence of the *attribute key* in product attribute collection as shown in diagram 3.5.

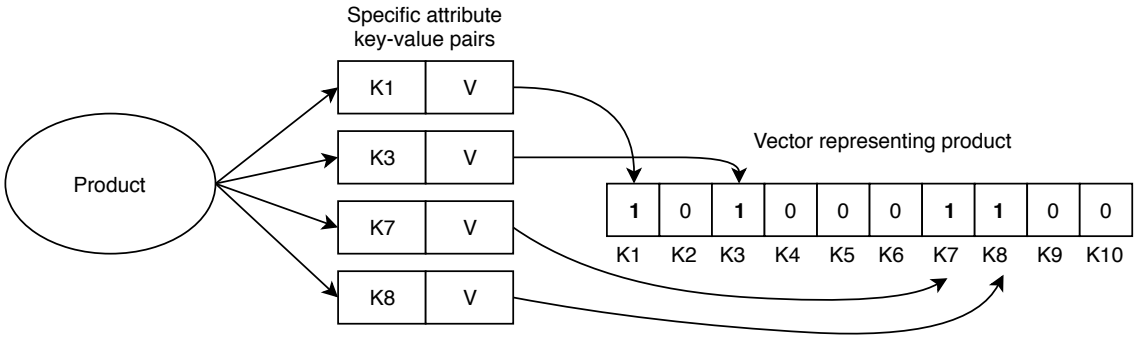



Figure 3.5: Product to vector diagram

**Property based similarity** To calculate property based similarity we need to represent product differently than just using attribute keys. We can base our product vector on attribute values. For string values we use *Levenshtein distance* and express the distance in interval $\langle 0.0, 1.0 \rangle$.

Numeric values are matched exactly. Error interval can be used, but it is difficult to decide the size of the interval because the value depends on the domain, unit and scenario.

This attribute value based vector representation can be used for clustering or general similarity calculation.

**Categorization** Clustering algorithms 2.3.2 can be used for category hierarchy building. With the vocabulary of existing categories, we can represent product by its connection with categories and then use the clustering algorithm to cluster products having similar categories. Furthermore, we can simply create category representatives and use any of the product vector representation to cluster similar products.

#### 3.2.2.2 Data transformation and reasoning

Since data are annotated and connections are made as described in previous section, we can integrate data and ontology together. Ontology individual creation can be broken down to multiple steps. These steps are described in following paragraphs. Individual *IRI* is based on the *ontology IRI* and the class type with unique integer ID. For example *IRI: https://dev.novotmike.com/oes#Product0* represents ontology with *IRI: https://dev.novotmike.com/oes*, class *Product* and $id = 0$. Diagram 3.6 shows processing diagram of the data transformation.

**Step 1: Loading ontology** Ontology OWL file as defined in 3.1.1 is loaded and classes, object properties and data properties are extracted and loaded.

**Step 2: Creating vocabulary individuals** As described in section 3.2.1.4. For each vocabulary entry, an individual is created and related class assigned. These individuals will be mapped to *products* using object properties from the ontology.

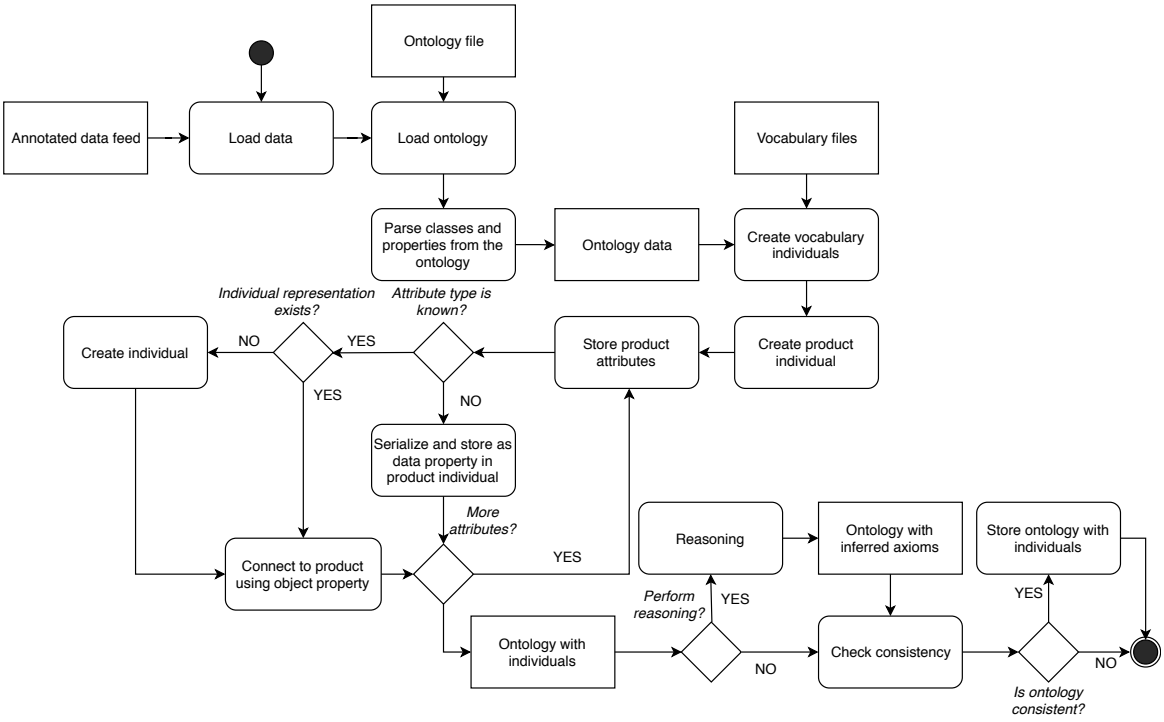



Figure 3.6: Ontology individuals creation processing diagram

**Step 3: Creating individuals for products** Each product individual is created from annotated data. Annotated common attributes are mostly represented in the ontology by class (price, manufacturer,...). These attribute individuals are created and connected based on the ontology.

Matched specific attributes with units are created as individuals and have a unit connected to them.

Rest of the attributes (unannotated) is stored within the product as a data property. Each property has *IRI* assigned based on the attribute type name and literal value is a CSV serialization of original *type name* and *value*. These attributes are stored within the product to make the reasoning fast and will be later extracted and stored as instances of *Attribute* class and connected to the product via *hasAttribute* relationship (see section 3.2.2.3).

Diagram 3.7 shows how resulting individual is represented in the ontology.

**Step 4: Product relation building** Using methods described in 3.2.2.1 we calculate the similarity between products. Similar products are connected using *isSimilarTo* object property. Additional relationships can be added to the ontology based on the specific attributes.

**Step 5: Consistency checking and reasoning** After all individuals have been created a reasoner is used to check whether the ontology is still consistent. If it is consistent, then reasoner runs over the ontology and infers new connection based on the property and *SWRL*

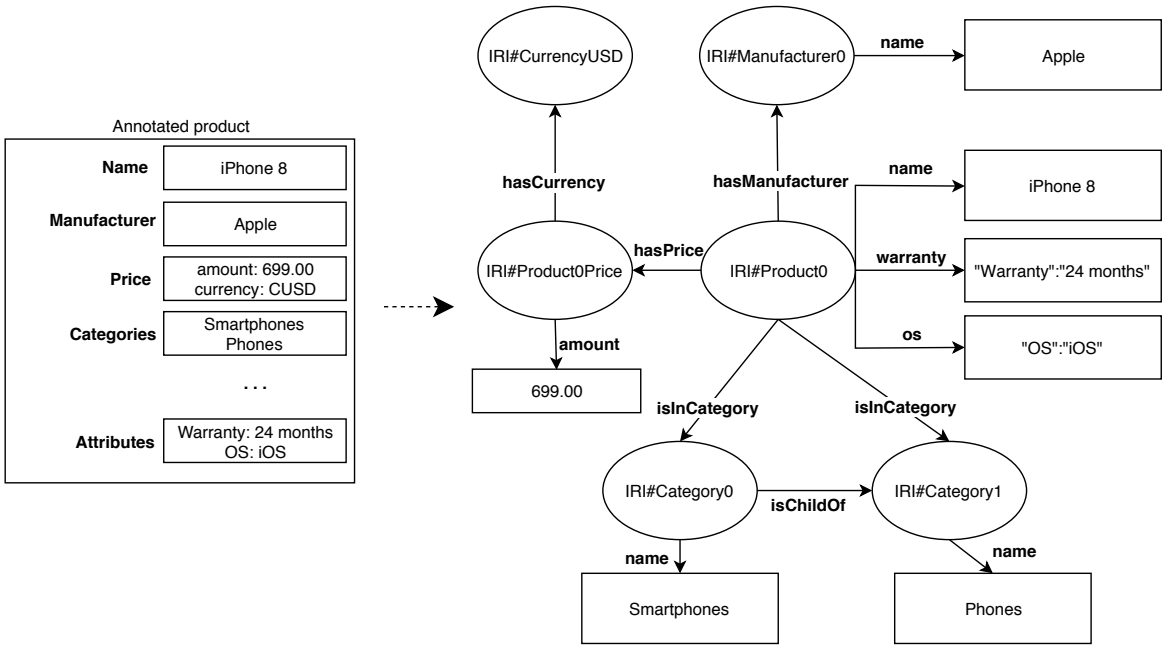



Figure 3.7: Diagram of ontology individuals created from annotated product data.

rules. Finalized ontology is stored into *RDF/XML* format and is ready to be transferred into the database storage.

#### 3.2.2.3 Storing data

Before the storage step data is in *RDF/XML* format stored along with the ontology. For data storage, we will be using a graph database. A major difference between *RDF triple store* and *labeled property graph* (LPG) is that LPG supports additional properties inside the relationships and also multiple connections between same nodes. The semantic data will be stored in the database in a way inspired by article [22].

We will use Neo4J [7] as our graph database. Each node is defined by a *label* which is a class in the graph database and is an equivalent of a class in our ontology.

The steps of the mapping process are described in the following list.

1. **Graph labels are defined from classes**
2. **Class individuals from RDF are created as nodes**

   For each individual in the ontology, we create a node with a label based on the *rdf:type* connection within the ontology. Each node is also defined by *UUID* (universally unique identifier) and the original *IRI* of the resource.

3. **Attribute individuals are clustered to create single node** As described in section 3.2.1.3 specific attributes with the matched unit are stored within the ontology as individuals. We create a node with *Attribute* label representing each unique specific

attribute and connect the unit to it. The product is connected to the attribute and the value is stored as a property of the relationship *hasAttribute*.

The same process applies for unmatched attributes contained as data properties within the individuals. The attribute is deserialized and a unique node is created for each attribute type. Value is stored in the relationship.

4. **Common attributes are stored as node properties** Basic common attributes of a product like *name* or *description* are stored as node properties.

5. **Predicates are stored as relationships**

   Remaining predicated - object properties - are stored as relationships between nodes which already exist in the database.

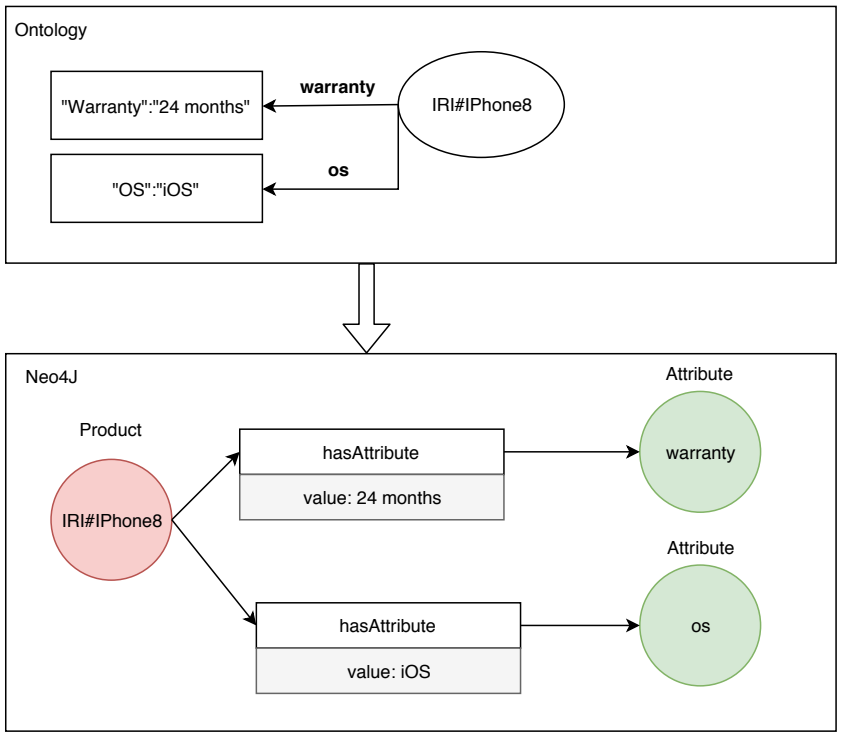



Figure 3.8: Specific attribute representation transformation.

### 3.2.3 Data querying

In previous sections we talked about *Feed library* and it's process, In section 3.2.2.3 we described how exactly data is stored, but now we have to define how queries going to be built. Since our data is semantic we know what exactly is stored and we can use this information for query building. We propose a simple system where the queries are built from ontology (3.1.1).

#### 3.2.3.1 Building the query

We will build queries in "triple-like" fashion: *Subject* → *Relation* → *Object* → *Value*. This means we are going to query specific class or property of the class and use already defined relationship types from the ontology (object properties).

A subject of the query is an *ontology class* associated with given node. It is what we want as a result. A relation is a object property from the ontology. An object of the query is a string representing data property of a node. *Value* is a string value of the node's data property. This simple pseudo-query language enables us to use ontology to query data in logical way. Further, we define an operator and allow for multiple relationships to be combined with and logic. This allows for more complex queries. Listing 3.3 shows how text queries can be translated into our pseudo-query. For purposes of this thesis, we will query the data using our defined pseudo-query language. Graphics representation of the pseudo-query is displayed in figure 3.9.

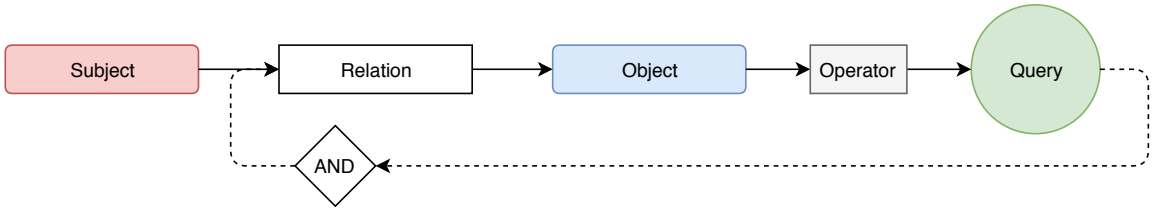



Figure 3.9: Pseudo-query graphical definition.

```
Q1: "Get all products similar to Macbook Pro"
Subject: Product
Relation: isSimilarTo
Object: name
Operator: =
Query: Macbook Pro

Q2: "Get all headphones under 500 CZK."
Subject: Product
Relation: isInCategory
Object: name
Operator =
Query: headphones
–AND Relation: hasPrice
– Object: amount
– Operator <
– Query: 500
```

Listing 3.3: Example pseudo-query representation

# Chapter 4

# Implementation

In the previous chapter we analyzed e-commerce data, discussed how to acquire knowledge from the data and what are the possibilities of storage. In addition, we designed an ontology 3.1.1 and a system which uses the ontology to transform e-commerce data and enhances it with knowledge. In this chapter, we will talk about how the designed solution was incorporated into a working system which allows users to query the stored semantic data.

## 4.1 System infrastructure

We already described *Feed library* (section 3.2) and how data is going to be stored and queries built. But this is just a part of the whole system. We need to be able to retrieve and visualize the data stored in the database in order to test the system. For purposes of this theses, we designed and implemented a REST (Representational State Transfer) [24] service which handles communication with the database and serves data to the client in a structured way. In addition, we created a simple client which allows user to input query and retrieve a list of results. All components of the system are noted in the following list.

- Feeder library
- API service
- Graph database
- Client

The whole system is deployed on the server as described in diagram 4.1 and 4.2. We used a *Forpsi Cloud VPS*[1] as our server. Parameters of the server are listed in table 4.1. Communication with the components is secured over *HTTPS* using certificates from *Let's Encrypt service* [6] and the server is available from a domain. In addition *NGINX* [8] server is used as *reverse-proxy* to manage communication with multiple components running on the same machine.

[1] https://www.forpsicloud.cz

| CPU | 1x Intel Xeon 1.80GHz |
|---|---|
| RAM | 2 GB |
| Disk space | 40 GB SSD |
| Operating system | Ubuntu Server 16.04 LTS 64bit |

Table 4.1: Server parameters.

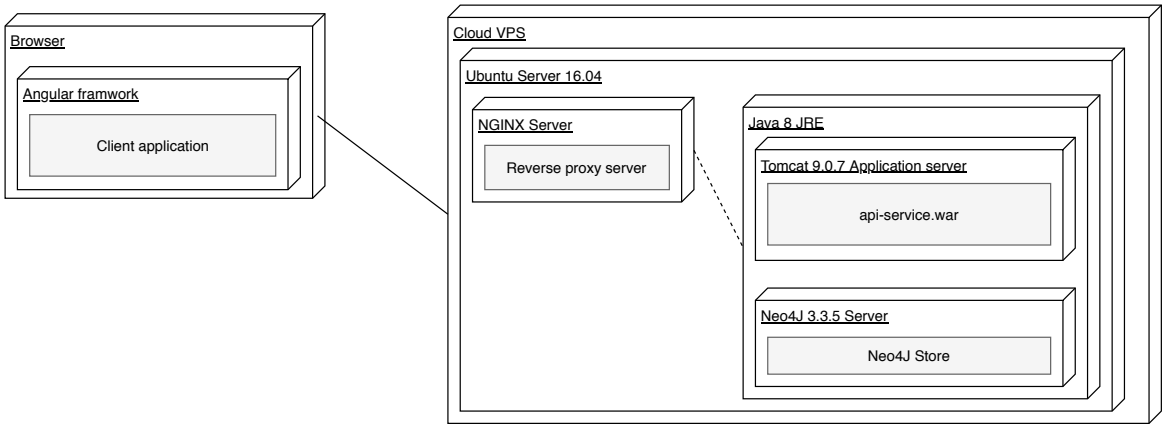



Figure 4.1: Deployment diagram of the system.

### 4.1.1 Tools and technologies

In this section, we will list all technologies used in the implementation of the system together with licenses. These technologies are summed in table 4.2.

| Name | Version | Description | License |
|---|---|---|---|
| Java JDK and JRE | 1.8 | Runtime environment | GNU GPL |
| Apache Tomcat | 9.0.8 | Application server | Apache License 2.0 |
| NGINX | 1.10.3 | Reverse proxy server | 2-clause BSD |
| Neo4J | 3.3.5 | Graph database | GNU GPL v3 |
| Angular | 5.2.9 | Javascript client framework | MIT License |
| Git | 2.16 | Versioning system | GNU GPL |
| Apache Maven | 3.3.9 | Java project management tool | Apache License 2.0 |
| Python | 3.6.5 | Programming language | PSF License |
| Beautiful Soup | 4.4.0 | HTML data extraction python library | MIT License |
| Hermit | 1.3.8.510 | An ontology reasoner | GNU GPL v3 |
| OWLAPI | 5.1.0 | Java library for ontology manipulation | Apache License 2.0 |
| Swagger | 2.0 | Tool for API documentation | Apache License 2.0 |

Table 4.2: Main component technologies used for implementation.

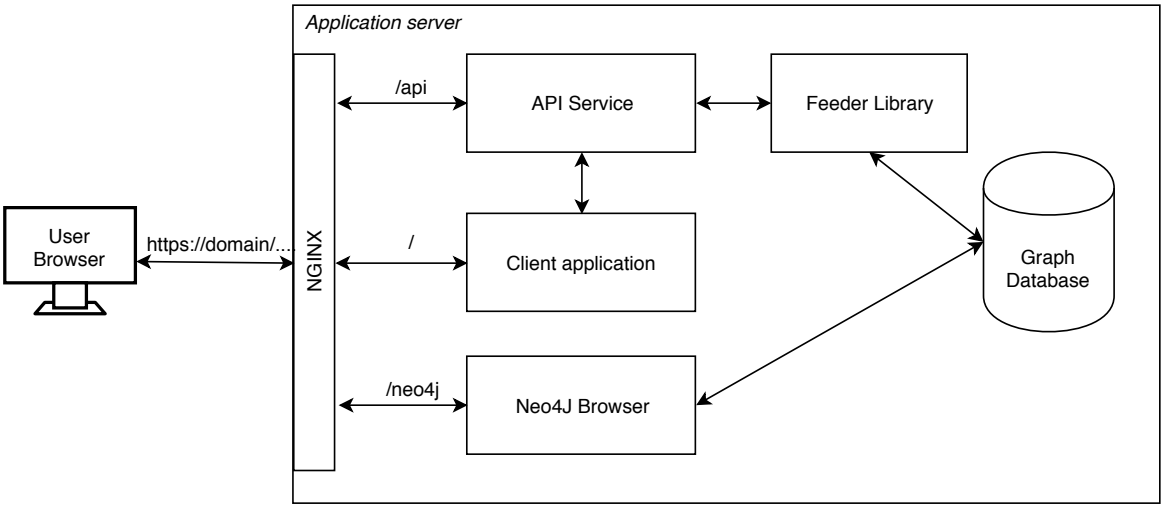



Figure 4.2: System architecture diagram.

### 4.1.2 Feeder library

We implemented *Feeder library* as it was described in section 3.2. We used *Java* language and *Maven* is used for project management. The library provides an interface for converting and annotating the data and for storing the data in the database.

#### 4.1.2.1 Library components

Main components and their classes are shown in diagram 4.3. As we described in previous chapter library can be divided into two main components *Feed component* and *Consolidation component*. Feed component is represented by *FeedDataService* class which loads key and value conversion maps, unit conversion map and manufacture vocabularies. It provides an interface for data feed management. Consolidation component is represented by *OWLService* and *GraphService* classes where *OWLService* is responsible for annotated data manipulation and ontology management. *GraphService* class provides interfaces for ontology storing into the graph database.

Both components are accessible over *FeederLib* class which acts as a facade of the library and provides base methods for data feed management, annotation, ontology creation and ontology storing. Feeder library further provides interface for *Neo4JSession factory* and *FeedStatistics* which are used from the *API Service*.

#### 4.1.2.2 Data meaning extraction

Using our conversion maps and vocabularies as described in design section 3.2.1.3 we annotated our data feeds listed in table 2.1. We extracted known attributes if they were present in the data. Some data did not contain all of the ontology defined classes.

Namely *Alza.cz* data did not contain data about *Manufacturers*, because the manufacturer was contained in the name of the product. Since the list of *Manufacturers* is one of our

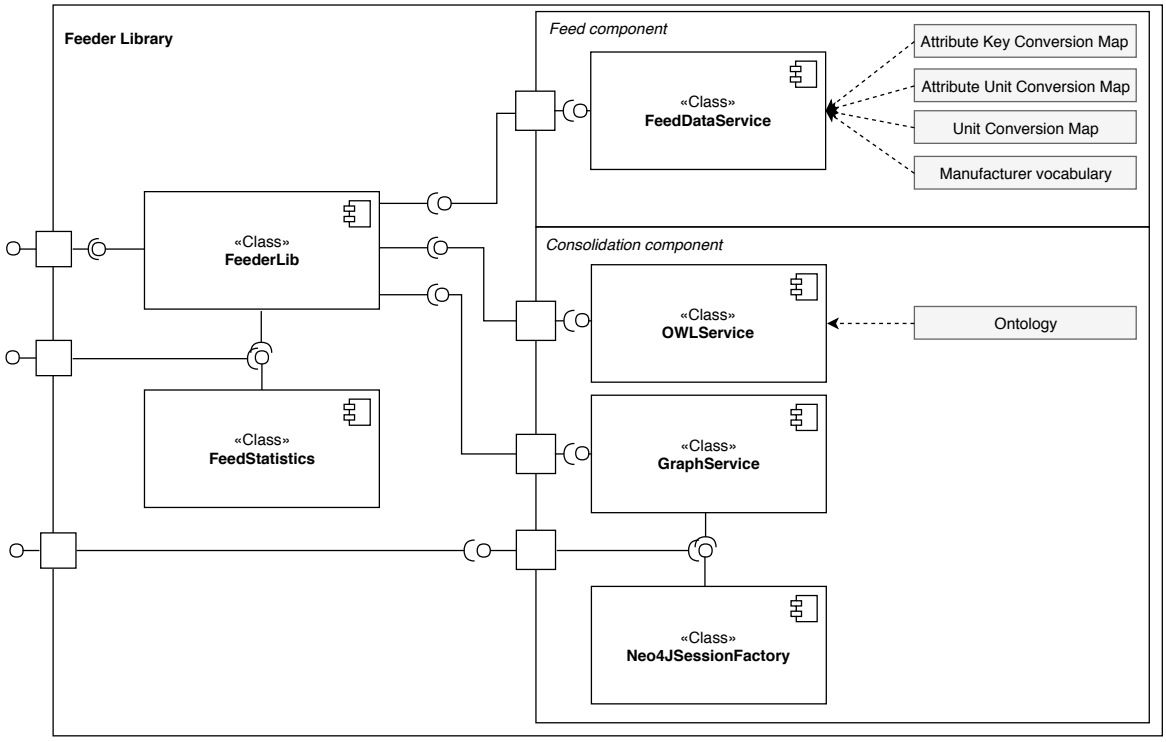



Figure 4.3: Feeder Library component and interface architecture diagram.

vocabularies we wanted to connect the product correctly. We used existing manufacturer vocabulary from the *Kasa.cz* feed and tried to infer manufacturers from the names of the products.

**Unit parsing programs** We defined interface for a unit parsing program and created a simple implementation for *amount-unit* pair extraction for *price*, *weight*, and *dimension* attribute types. Furthermore general implementation of this program was used to extract *amount-unit* pair from other attributes.

Diagram 4.4 displays *amount-unit* parser implementations and the factory class which loads parser implementation based on the attribute type. We implemented parsers for *price-currency* pair and general *amount-unit* parser. Both parsers are based on *regular expressions*.

- Price-currency regex pattern: **([$L€])?([ ]*)([0-9.,]+)([ ]*)([a-zA-Zè$L€]+)?**
- Amount-unit regex pattern **(^[0-9., ]+)([ ]*)([a-zA-Z]+)**

#### 4.1.2.3 Ontology building and data storage

Annotated data are loaded using *Feeder library* and the ontology is loaded and parsed using *OWL API Java* library. Loaded classes, object and data properties are used to map annotated product into an ontology individual as described in section 3.2.2.2.

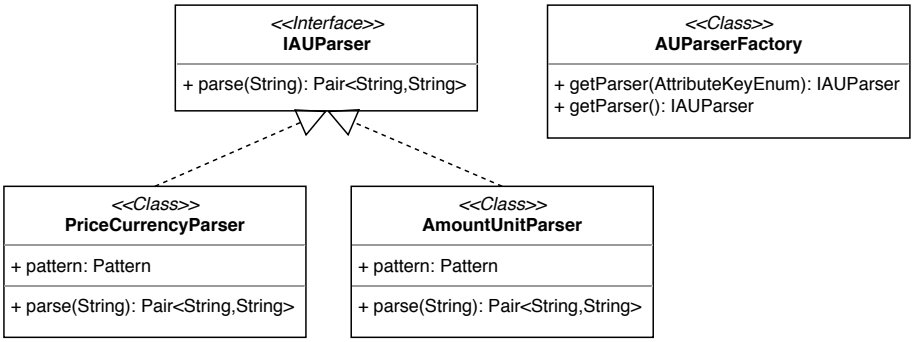



Figure 4.4: Amount unit parsing factory class diagram.

**Relationship building** After all products have been transformed into ontology individuals, relation specific programs calculate relationships between products and categories. We defined *relationship builder* interface and implemented classes for product *similarity* and *compatibility* calculation. Further more *relationship builder* was also defined for category hierarchy building. Figure 4.5 shows class diagram of product *relationship builder* factory a and implementations.

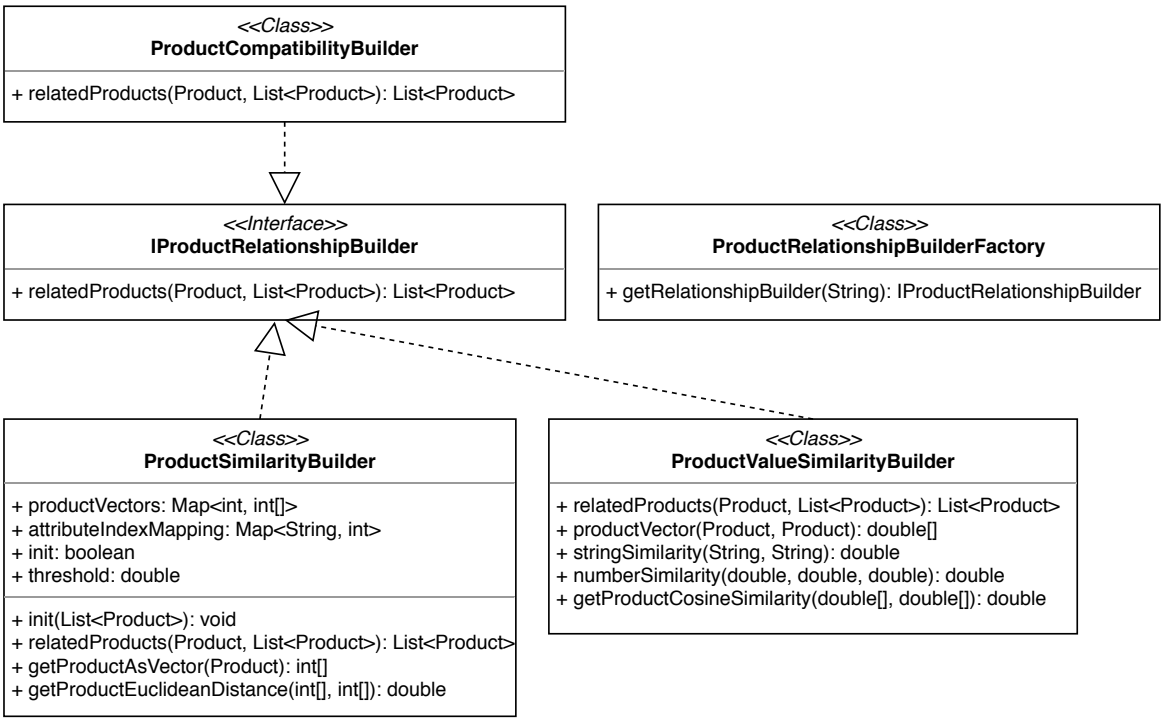



Figure 4.5: Product relationship builder class diagram.

Relationship builders are based on methods discussed in 3.2.2.1. *ProductSimilarityBuilder* and *ProductValuesSimilarityBuilder* are used to create *isSimilarTo* connection. *ProductCom-*

*patibilityBuilder* creates *isCompatibleWith* relationship between products.

**Reasoning and consistency checking** Next we used *Hermit* reasoner to check consistency of the final ontology and for inferring new relationships. The reasoner infers mainly SWRL rule for *category containing product* (rule 2.2) and inverse object property connections. Finalized ontology of the annotated data sample is stored in the *RDF/XML* format. This data was then stored into *Neo4J* database (as described in 3.2.2.3).

**Neo4J browser** Neo4J database comes with *Neo4J browser* application which allows us to test *CYPHER* queries. It also provides graph visualization tools. Figure 4.6 shows graph visualization of product, price and currency connection.

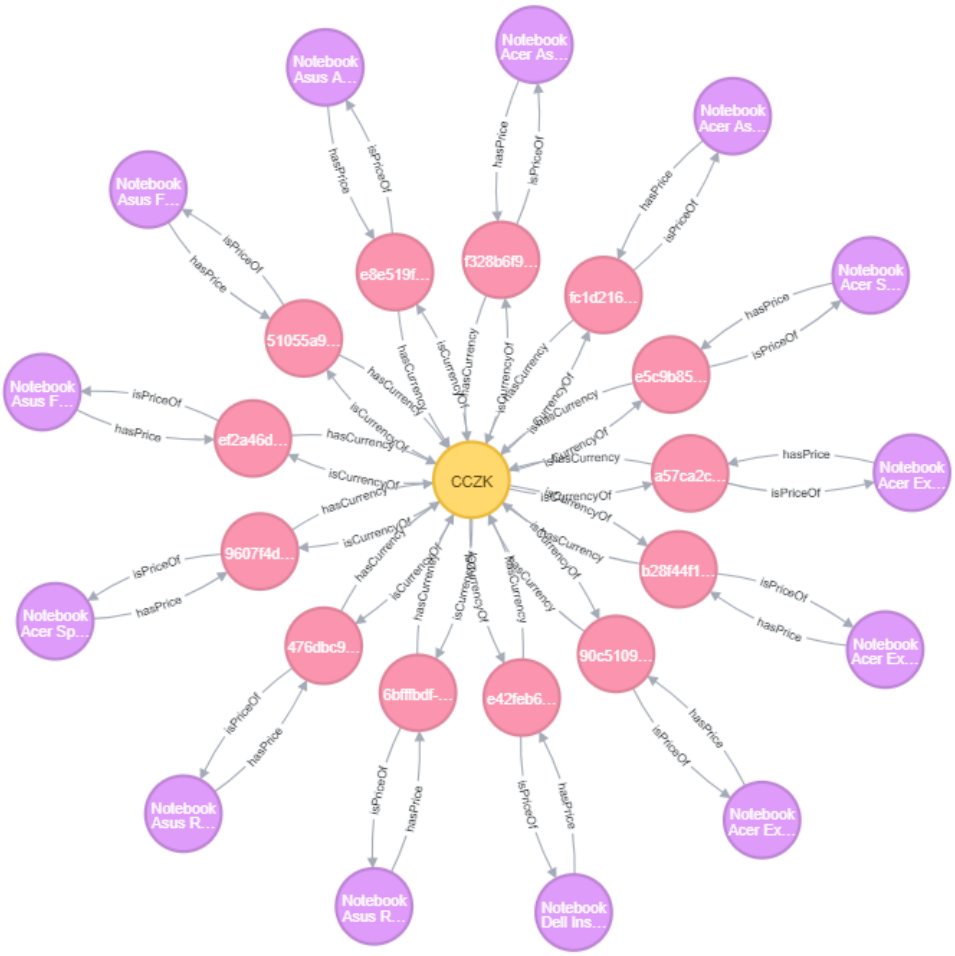



Figure 4.6: Product, price and currency connection in Neo4J database.

### 4.1.3 API service

We implemented a service which is responsible for handling data queries over the stored semantic data. Endpoints of the service are defined in table 4.3 and their documentation in *Swagger 2.0*[18] format can be found attached in the appendix B.

We used *Java* programming language together with *Maven* and other libraries to build our API service. We chose *Java* because it is a standard technology for web application and in addition the *Neo4J* database system is also implemented using *Java* which enables good integration.

| Path | Description |
|---|---|
| /query | Endpoint for search queries in format defined in 3.2.3.1 |
| /query/options | Endpoint for getting all the class and relationship types based on the ontology and stored data |

Table 4.3: Application service endpoints

API service uses *Feeder library* interface for obtaining database connection and data from the ontology for query building.

#### 4.1.3.1 Cypher query building

In order to query the data, we need to translate our defined pseudo-query (see section 3.2.3.1) into Neo4J query language Cypher. Since our pseudo-query is defined from ontology and stored data also copy ontology structure we can directly interpret cypher query from our pseudo-query. Query building relies on unified units in the stored data.

**Attribute values query** Only exception for straightforward query generation are *Attribute* labeled nodes with values stored in their *hasAttribute* relationship. For this reason we treat attribute relationship differently and allow for two types of properties: *class properties* and *relationship properties*.

Listing 4.1 shows examples pseudo-query to cypher translation. One of the example contains the *hasAttribute* connection and one is normal.

```
Q1: "Get all products similar to Macbook Pro"
Subject: Product
Relation: isSimilarTo
Object: name
Operator: =
Query: Macbook Pro

Cypher: MATCH (p:Product)-[:isSimilarTo]-(o)
        WHERE o.name = 'Macbook Pro'
        RETURN p
```

```
Q2: "Get all products which have more than 4000 MB RAM"
Subject: Product
Relation: hasAttribute
Object: RAM
Operator >
Query: 4000

Cypher: MATCH (p:Product)-[r:hasAttribute]-(o)
        WHERE r.val > 4000 AND o.name='RAM'
        RETURN p
```

Listing 4.1: Example pseudo-query representation

### 4.1.4 Client application

Client application was implemented using *Angular* javascript framework. We chose *Angular* because it is a standard technology for the creation of web clients for RESTful applications. The client is a *single-page* MVC (Model View Controller) application which displays query results and contains a *form* for creation of the pseudo-queries.

Client serves as a tool to test the pseudo-query definition and for visualizing the data. Even though Neo4J offers *browser* application for visualization we implemented the client to complete the system and test the API service.

**API communication** Classes and relationships for the query building are requested from the API service. Communication between client and API service is documented in a API documentation in appendix section B. Figure 4.8 shows implemented client user interface. It enabled user to build a *pseudo-query* using the options from the *API Service*.

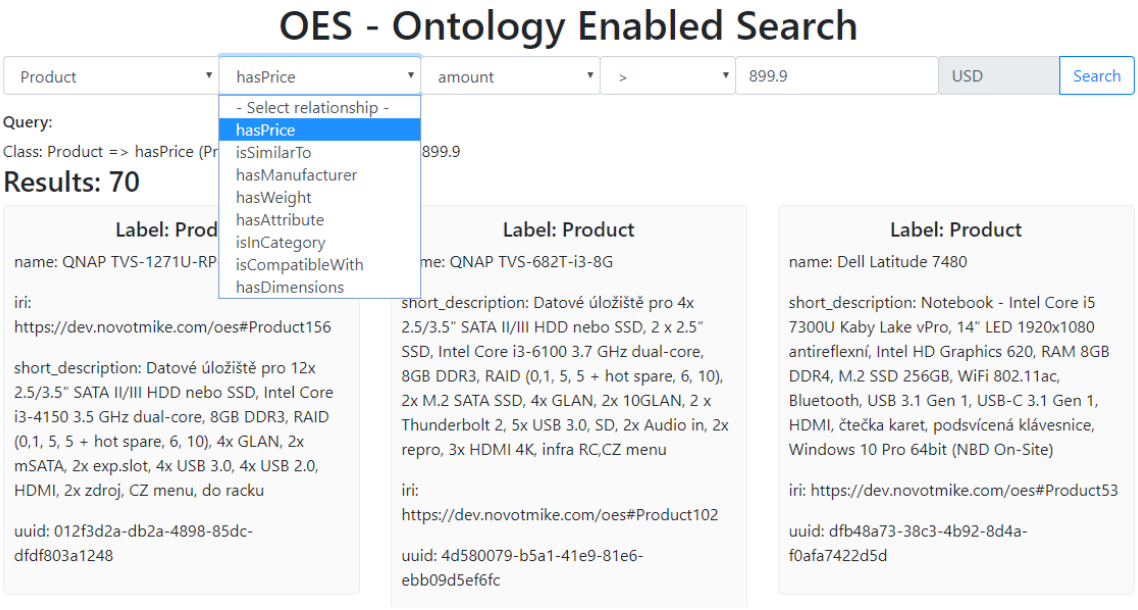



Figure 4.7: Client application query building screenshot

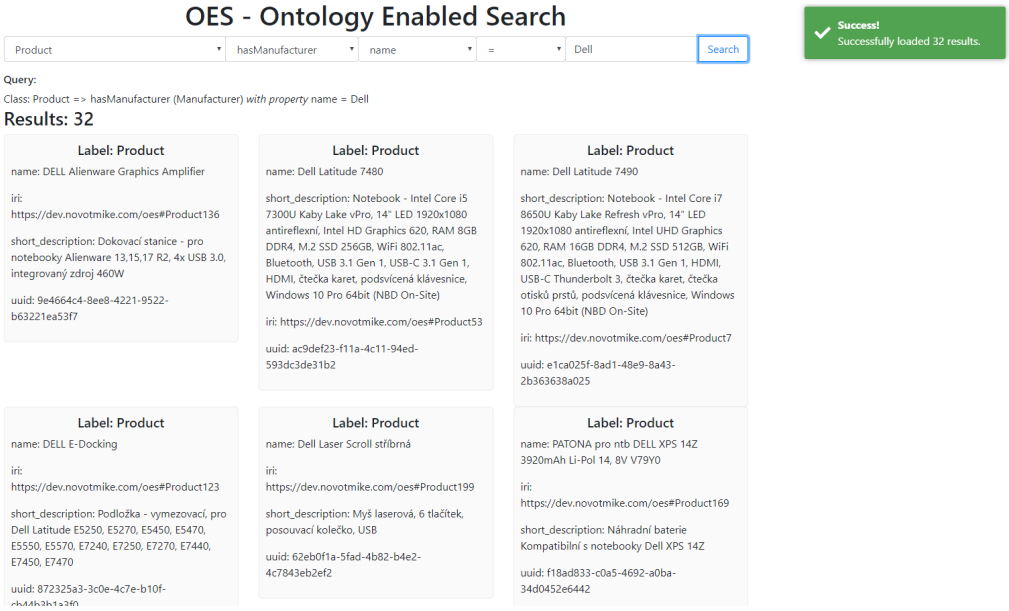



Figure 4.8: Client application screenshot

# Chapter 5

# Evaluation

In this chapter, we will talk about how we evaluated the solution design and the final implementation. We created a raw data sample of various products which we are going to use for the testing. We also created a set of products which we annotated in a way that will enable us to test the search and similarity matching.

## 5.1 Data evaluation

We used our implemented *Feeder library* to annotate the data feeds (listed in table 2.1). As a first step, we created general statistics from the annotated data feeds. Results are shown in table 5.1.

| | |
|---|---|
| **Number of products** | 46613 |
| **Number of attribute keys** | 705 |
| **Number of manufacturers** | 1129 |
| **Number of categories** | 1772 |
| **Average number of attributes** | 15.41 |
| **Average percentage of matched attributes** | 52% |
| ***Name* attribute percentage** | 100% |
| ***Price* attribute percentage** | 100% |
| ***Manufacturer* attribute percentage** | 72% |
| ***Category* attribute percentage** | 97% |
| ***Description* attribute percentage** | 36% |
| ***Dimensions* attribute percentage** | 15% |
| ***Weight* attribute percentage** | 13% |

Table 5.1: General statistics generated from all annotated data feeds.

The table 5.1 describes basic statistics of all data feeds combined. *Average number of attributes* states how many attributes has a product on average. *Average percentage of matched attributes* states the percentage amount of how many attributes from all attributes were matched and annotated with meaning. Additional percentages state how each common attribute is matched within the data. We can see that *name* and *price* attributes were

matched on all of our products. Other attributes such as *dimensions* or *weight* are not as common. This is due to the fact that the non-scraped data feeds do not contain these values.

### 5.1.1 Testing data

For our testing scenarios, we created few datasets to test and evaluate how data annotation and ontology building works. Each dataset was created from the scraped data or existing data feeds. Products were chosen randomly and in some cases domain specific. Testing data is represented in *unified feed format* (3.1.2). Table 5.2 describes created datasets and their sources.

| | Name | **Domain** | **Source** |
|---|---|---|---|
| 1. | TEST1000 | Phones, Phone accessories | *Alza.cz* feed |
| 2. | TEST2000 | Notebooks, Notebook accessories | *Alza.cz* and *Kasa.cz* feeds |
| 3. | TEST5000 | Clothing and outdoor equipment | *4Camping* data feed |
| 4. | TEST100x100 | Notebooks, Notebook accessories | *Alza.cz* data feed |

Table 5.2: Testing data overview

#### 5.1.1.1 User annotated data

In addition to our testing data, we also created a user annotated feed which will be used as a *ground truth* to evaluate how well the annotation process and searching works. The user annotated data feed was created from the *TEST100x100* data. It contains *100* products from notebook categories and *100* notebook accessories.

Users annotated basic properties with regards to our defined ontology. The following list sums all properties which were annotated.

- Name
- Manufacturer
- Categories
- Price with currency
- Weight with unit
- Dimensions with unit (width, height, depth)

Furthermore, products relationships were annotated as well. Users annotated the *isSimilarTo* property and *isCompatibleWith*. Using this annotated data we can evaluate how well relationship building algorithms work as well as how good the automatic annotation process is.

### 5.1.2 Testing scenarios

To evaluate our solution we defined three different testing scenarios for our data. Each scenario is described in following subsections. We used user annotated data set as well as the data sets we discussed before.

**Testing setup** As we described in section 4.1 our solution is designed to run on a server with parameters stated in table 4.1. Our database system and API service together with client run on the described server.

#### 5.1.2.1 I. Data annotation, ontology building and storing

In this scenario, we will evaluate how well the data annotation process works. We will list statistics for each dataset. We will also evaluate how fast each step of the transformation is with regards to each data set, the number of attributes and the ontology.

**Data statistics** For each dataset, after annotation, we created a statistics file containing some basic information. These statistics can be seen in table 5.3. We can see that that *name*, *price* and *category* are attributes which are contained within almost every product. Furthermore we can see that in datasets *TEST1000* and *TEST2000* more than half of the products contain dimension and weight values. Further, we can see that additional *attribute keys with units* were found in both datasets.

The *TEST5000* dataset of clothing and outdoor equipment does not contain *dimension* or *weight* attributes but the *average percentage of matched attributes* is high. This is because the data contains almost exclusively common attributes (see table 2.2).

| | **TEST1000** | **TEST2000** | **TEST5000** |
|---|---|---|---|
| **Number of products** | 1000 | 2000 | 5000 |
| **Number of attributes** | 34311 | 78815 | 71450 |
| **Distinct attribute keys** | 139 | 348 | 11 |
| **Number of attribute keys with units** | 24 | 40 | 0 |
| **Number of manufacturers** | 67 | 103 | 166 |
| **Number of categories** | 110 | 160 | 178 |
| **Average number of attributes** | 28.9 | 32.4 | 11.0 |
| **Average percentage of matched attributes** | 32% | 32% | 74% |
| ***Name* attribute percentage** | 100% | 100% | 100% |
| ***Price* attribute percentage** | 100% | 100% | 100% |
| ***Manufacturer* attribute percentage** | 84% | 95% | 99% |
| ***Category* attribute percentage** | 100% | 100% | 94% |
| ***Description* attribute percentage** | 0% | 50% | 0% |
| ***Dimensions* attribute percentage** | 54% | 63% | 0% |
| ***Weight* attribute percentage** | 53% | 54% | 0% |

Table 5.3: General statistics for each dataset.

**Data transformation** For each data set we performed the whole transformation from raw data in the unified data feed to the stored data in the database. We performed this test *10* times for each dataset and in the table 5.4 are listed average time values of each step from all the test runs.

| | TEST1000 | TEST2000 | TEST5000 |
|---|---|---|---|
| **Data annotation** | 0.67 s | 1.15 s | 0.34 s |
| **Ontology building** | 5.29 s | 5.78 s | 14.13 s |
| **Reasoning** | 277.4 s | 1077.1 s | 833.8 s |
| **Ontology storage** | 4.63 s | 4.98 s | 3.34 s |
| **Database storage** | 2260.2 s | 3807.9 s | 1795.0 s |

Table 5.4: Data transformation steps and their time.

From our time results we can see that the biggest bottleneck of the data to ontology transformation is *reasoning*. Reasoners infer new connections within the ontology and apply *SWRL* rules. The *property axiom generation* (inferring new properties) is the most time-consuming task. We can improve the reasoning calculation time by removing some inverse object properties and stripping relationships such as *isSimilarTo* of its *symmetry* property. This is due to the fact, that *Neo4J* does not care about the relationship direction and we do not need the inverse properties. In case of RDF storage though, we would loose some important relationships.

In the end the database storage is most time-consuming operation of all listed in table 5.4. It is that way due to the implementation and heavy use of transactions. In table 5.5 are shown node and relationship amounts. Data storage efficiency was not primary focus of this thesis and there is a room for improvement. A possibility is to use *stored procedures* in *Neo4J* or *Neo4J Java* library to make the storage into the database more efficient.

#### 5.1.2.2 II: Ontology evaluation

In this evaluation scenario we look into the generated ontology and how it looks and how it was extended during the data transformation. Having data individuals along the ontology definition allows us to perform consistency checking and reasoning. Base ontology has *11* classes, *18* object properties and *11* data properties defined. Table 5.5 shows quantification of each dataset ontology and the graph representation.

| | TEST1000 | TEST2000 | TEST5000 |
|---|---|---|---|
| **Classes** | 12 | 12 | 12 |
| **Object properties** | 18 | 18 | 18 |
| **Data properties** | 150 | 358 | 13 |
| **Individuals** | 6143 | 14445 | 10345 |
| **Neo4j Nodes** | 3432 | 7005 | 10347 |
| **Neo4j Relationships** | 99722 | 167868 | 71426 |

Table 5.5: Summary of data store in ontology and Neo4J.

We can see that during the ontology creation new data properties were defined. We described this in section 3.2.2.2. These data properties are later transformed into *Attribute* nodes in the database. We can also see that the number of nodes in the database is lower than number of individuals. This is due to the fact, that some *Attribute* individuals get merged together in the database to interconnect products with same attributes as describe in section 3.2.2.3.

**Stored data scheme** In figure 5.1 is displayed database scheme from the stored data. Although Neo4j database does not have a schema definition, the schema is generated from the stored data. We can clearly see how the schema copies our defined ontology.


Figure 5.1: Neo4J stored data scheme

#### 5.1.2.3 III: Relationship building evaluation

For further evaluation we will use our user annotated dataset. We were able to test *isSimilarTo* and *isCompatibleWith* relationship building. Results of the evaluation are stated in table 5.6.

| | **Precision** | **Recall** |
|---|---|---|
| **Similarity relation matching** | 39 % | 37% |
| **Compatibility relation matching** | 25% | 31% |

Table 5.6: General statistics for each dataset.

In the table 5.6 we can see that the precision of the similarity and compatibility relationships is not very high. This is due to the *domain specificity* of the data. Our ontology and the whole system is built non-domain specific. While it is an advantage when the overall performance is important, some specific data connection will be missed using our defined system.

#### 5.1.2.4 IV. Stored data evaluation

For evaluation of search queries, we used our client and API service. We stored the unannotated version of our user annotated data in order to be able to evaluate the results. In previous section, we evaluated how well relationships were mapped.

| | **TEST100x100** | **Match score** |
|---|---|---|
| **Name attribute percentage** | 100% | 100% |
| **Manufacturer attribute percentage** | 90% | 75% |
| **Price attribute percentage** | 100% | 100% |
| **Dimensions with unit attribute percentage** | 84% | 92% |
| **Weight with unit attribute percentage** | 65% | 98% |

Table 5.7: Solution annotated data and user annotated data comparison.

Since our search results rely heavily on the data meaning extracted from the data, we compared the data extraction against the user annotated data. Table 5.7 contain statistic information of how the *Feeder library* match the data and then how accurate are the results compared to the *ground truth*. The *TEST100x100* column describes the percentage of extracted attributes by the *Feeder library*. The *Match score* describes the percentage of correctly extracted data.

We can use our client and API service implementation to retrieve stored results. The resulting data precision is based on the matched attributes and precision of the relationships described in previous paragraphs and sections.

# Chapter 6

# Conclusion

This chapter discusses achieved results during the project and further steps which could be made to improve the system.

## 6.1 Summary

We were able to analyze current trends, method and tools for storing and representing e-commerce data. We talked about different ways how we can look at the data and proposed a simplified view of the products as a *collection of properties*. Next, we analyzed methods for data meaning extraction and knowledge representation using ontologies. We described tools and methods for ontology definition and why ontologies are a key component of semantic data and their storage. With ontology in mind, we analyzed existing data feeds and scraped data from existing e-commerce websites. We proposed a unified feed structure taking advantage of the simplified product representation.

We then analyzed data feeds and scraped data to design a simple ontology which we would use to store and represent semantic e-commerce data. We design a system which takes in data in the unified format and annotates them using various methods and programs using our designed ontology. We defined which attributes of the product are *common* and should be contained within the product data. We also analyzed methods for product similarity factor calculation.

Next, we designed a system which takes the annotated data and creates their representation inside our designed ontology. We then performed reasoning over the ontology to infer additional properties. Then a process of storing the data inside a graph database was designed. We implemented the defined system which finalized the data transformation from raw feed data into the semantically annotated data stored inside a graph database.

Furthermore, an API service and client together with simple query language were designed and implemented to simulate complete solution and to enable testing of the stored data and meaning extraction.

We evaluated the annotation process and resulting ontology and stored data. We were able to conclude that the annotation process relies on the analysis of the data and the data domain.

## 6.2 Further steps

In future, we would like to focus on expanding the ontology dynamically based on the domain. We would like to make use of the clustering algorithms to better infer product relationships. We would also want to use the system as a part of an existing e-commerce solution.

Other possible steps are pseudo-query building from user-defined text queries. This would enable the system to be used as a search engine and make use of the additional product and relationship information.

# Bibliography


[1] Apify website. Available from: <https://www.apify.com/>.

[2] Ecommerce Foundation Report webpage. Available from: <http://www.ecommercefoundation.org/reports>.

[3] FaCT++ website. Available from: <http://owl.cs.manchester.ac.uk/tools/fact/>.

[4] Hermit website. Available from: <http://www.hermit-reasoner.com/>.

[5] Heureka website. Available from: <https://www.heureka.cz>.

[6] Let's Encrypt website. Available from: <https://letsencrypt.org>.

[7] Neo4J website. Available from: <https://neo4j.com/>.

[8] NGINX wiki. Available from: <https://www.nginx.com/resources/wiki>.

[9] OWL API website. Available from: <http://owlcs.github.io/owlapi/>.

[10] Pellet website. Available from: <https://github.com/stardog-union/pellet>.

[11] PostgreSQL website. Available from: <https://www.postgresql.org/>.

[12] Protégé website. Available from: <https://protege.stanford.edu/>.

[13] Wiki: RDF - Resource Description Framework. Available from: <https://en.wikipedia.org/wiki/Resource_Description_Framework>.

[14] Redis website. Available from: <https://redis.io/>.

[15] Google RSS 2.0 documentation. Available from: <https://cyber.harvard.edu/rss/rss.html>.

[16] Scrapy website. Available from: <https://scrapy.org/>.

[17] ShopAPI website. Available from: <https://shopapi.cz/>.

[18] Swagger website and documentation. Available from: <https://swagger.io/>.

[19] Wikipedia. Available from: <https://en.wikipedia.org>.

[20] YAML website and documentation. Available from: <http://yaml.org/>.

[21] BARRASA, J. Neo4J Article, . Available from: <`https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/`>.

[22] BARRASA, J. Importing RDF data into Neo4J an Article, . Available from: <`https://jbarrasa.com/2016/06/07/importing-rdf-data-into-neo4j/`>.

[23] CONSORTIUM, W. W. W. Resource Description Framework. Available from: <`:http://www.w3.org/2004/OWL`>.

[24] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[25] GRUBER, T. What is an ontology? Available from: <`http://www-ksl.stanford.edu/kst/what-is-an-ontology.html`>.

[26] NOY, N. F. – MCGUINNESS, D. L. Ontology Development 101: A Guide to Creating Your First Ontology. Technical report, 2001.

[27] O'REILLY, T. *What is Web 2.0?* 2005.

[28] ROWLEY, J. The wisdom hierarchy: representations of the DIKW hierarchy. *Information and Communication Science*. 2007, s. 163–180. doi: doi:10.1177/0165551506070706.

[29] ROWLEY, J. The wisdom hierarchy: representations of the DIKW hierarchy. *Journal of Information Science*. 2007, 33, 2, s. 163–180. doi: 10.1177/0165551506070706. Available from: <`https://doi.org/10.1177/0165551506070706`>.

[30] TIM BERNERS-LEE, J. H. – LASSILA, O. The Semantic Web. 2001.

[31] TUDORICA, B. G. – BUCUR, C. *A comparison between several NoSQL databases with comments and notes*. IEEE, Jun 2011. doi: 10.1109/roedunet.2011.5993686. Available from: <`http://dx.doi.org/10.1109/RoEduNet.2011.5993686`>.

[32] WALLACE, D. P. Knowledge Management: Historical and Cross-Disciplinary Themes. *Libraries Unlimited*. 2007, s. 1–14.

[33] XU, R. – WUNSCH, D. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*. May 2005, 16, 3, s. 645–678. ISSN 1045-9227. doi: 10.1109/TNN.2005.845141.

# Appendix A

# Nomenclature

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

CAP Consistency, Availability, Partition tolerance

CPU Central Processing Unit

CSV Comma-Separated Values

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IRI Internationalized Resource Identifier

JSON JavaScript Object Notation

LPG Labeled Property Graph

MVC Model View Controller

OWL Web Ontology Language

RAM Random-Access Memory

RDF Resource Description Framework

REST Representational State Transfer

RSS Rich Site Summary

SPARQL SPARQL Protocol and RDF Query Language

SQL Structured Query Language

SSD Solid-State Drive

SWRL Semantic Web Rule Language

URI Uniform Resource Identifier

UUID Universally unique identifier

VPS Virtual Private Server

XML Extensible Markup Language

YAML YAML Ain't Markup Language

# Appendix B

# API service documentation

The following paragraph contains API service documentation in *Swagger 2.0* format (serialized in YAML).

### Documentation

```yaml
---
swagger: "2.0"
info:
  description: This is a simple API
  version: 1.0.0
  title: Ontology enabled search API
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
host: virtserver.swaggerhub.com
basePath: /michal.novotny/OES_API/1.0.0
tags:
- name: user
  description: System users
schemes:
- https
paths:
  /query:
    post:
      tags:
      - user
      summary: Queries the data
      description: Queries the stored data based on the specified OQuery
      operationId: queryData
      consumes:
      - application/json
      produces:
```

```
    - application/json
    parameters:
    - in: body
      name: query
      description: Query to be used
      required: true
      schema:
        $ref: '#/definitions/OQuery'
    - name: X-Limit
      in: header
      description: Limit of the resulting data
      required: false
      type: integer
    - name: X-Offset
      in: header
      description: Offset of the resulting data
      required: false
      type: integer
    responses:
      200:
        description: Result of the search query
        schema:
          $ref: '#/definitions/GraphResultsDTO'
      400:
        description: Bad request - invalid query
        schema:
          $ref: '#/definitions/Message'
      500:
        description: Internal server error
        schema:
          $ref: '#/definitions/Message'
/query/options:
  get:
    tags:
    - user
    summary: Get query options from ontology
    operationId: getQueryOptions
    consumes:
    - application/json
    produces:
    - application/json
    parameters: []
    responses:
      200:
        description: Query options
        schema:
```

```
            $ref: '#/definitions/QueryOptionsDTO'
        404:
          description: Ontology could not be loaded
          schema:
            $ref: '#/definitions/Message'
        500:
          description: Internal server error
          schema:
            $ref: '#/definitions/Message'
definitions:
  PropertyDTO:
    type: object
    properties:
      name:
        type: string
        example: RAM
      unit:
        type: string
        example: Gigabyte
    example:
      unit: Gigabyte
      name: RAM
  RelationshipDTO:
    type: object
    properties:
      name:
        type: string
        example: isSimilarTo
      range:
        type: array
        items:
          type: string
    example:
      name: isSimilarTo
      range:
      - range
  ClassDTO:
    type: object
    properties:
      name:
        type: string
        example: Product
      relationships:
        type: array
        items:
          $ref: '#/definitions/RelationshipDTO'
```

```
    properties:
      type: array
      items:
        $ref: '#/definitions/PropertyDTO'
  example:
    relationships:
    - name: isSimilarTo
      range:
      - range
      - range
    name: Product
    properties:
    - unit: Gigabyte
      name: RAM
QueryOptionsDTO:
  type: object
  properties:
    classes:
      type: array
      items:
        $ref: '#/definitions/ClassDTO'
  example:
    classes:
    - relationships:
      - name: isSimilarTo
        range:
        - range
        - range
      name: Product
      properties:
      - unit: Gigabyte
        name: RAM
Message:
  type: object
  properties:
    message:
      type: string
      example: Some API message
OQuery:
  type: object
  properties:
    subject:
      type: string
      example: Product
    relationships:
      type: array
```

```
      items:
        $ref: '#/definitions/ORelationship'
  example:
    relationships:
    - query:
        value: Apple
        operator: EQ
      name: hasManufacturer
      type: CLASS_PROP
      object: name
    subject: Product
ORelationship:
  type: object
  properties:
    type:
      type: string
      example: CLASS_PROP
    name:
      type: string
      example: hasManufacturer
    object:
      type: string
      example: name
    query:
      $ref: '#/definitions/OQueryValue'
  example:
    query:
      value: Apple
      operator: EQ
    name: hasManufacturer
    type: CLASS_PROP
    object: name
OQueryValue:
  type: object
  properties:
    operator:
      type: string
      example: EQ
    value:
      type: string
      example: Apple
  example:
    value: Apple
    operator: EQ
GraphResultsDTO:
  type: object
```

```
    properties:
      results:
        type: array
        items:
          $ref: '#/definitions/GraphResultDTO'
    example:
      results:
      - properties:
          name: Apple iPhone
          uuid: 0c76c148-919f-4337-a05c-9e608b4df184
        labels:
        - Product
  GraphResultDTO:
    type: object
    properties:
      labels:
        type: array
        items:
          type: string
          example: Product
      properties:
        type: object
        example:
          name: Apple iPhone
          uuid: 0c76c148-919f-4337-a05c-9e608b4df184
        additionalProperties:
          type: string
    example:
      properties:
        name: Apple iPhone
        uuid: 0c76c148-919f-4337-a05c-9e608b4df184
      labels:
      - Product
```

# Appendix C

# Contents of attached CD

- Folder: *text/*
  - NovotnyMichal_MT.pdf - PDF version of the Master's thesis
- Folder: *implementation/*
  - *server/* - Folder containing server source code
  - *client/* - Folder containing client source code
  - *feeder/* - Folder containing feeder library source and ontology data
    - *ontology/* - Folder containing all ontology versions and generated ontologies
    - *input/* - Folder containing all input feeds and annotated data
    - *scraping/* - Folder containing scraping python programs
  - README.txt - file containing compilation and other information
- File: *Documentation.pdf* - file containing basic information about implementation.