Federated Product Information Search and Semantic Product Comparisons on the Web

Dissertation
submitted in partial satisfaction of the requirements
for the degree of Doktoringenieur
(Dr.-Ing.)

at
Technische Universität Dresden
Faculty of Computer Science

by
Dipl.-Medieninf. Maximilian Thilo Walther
born on May 5th, 1982 in Darmstadt

Advisers:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill TU Dresden
Prof. Dr. rer. nat. Claudia Linnhoff-Popien LMU München
Prof. Dr.-Ing. Michael Schroeder TU Dresden

Date of Defense: September 9, 2011
Dresden, September 2011
Confirmation

Herewith I confirm that the present dissertation about Federated Product Information Search and Semantic Product Comparisons on the Web has been prepared independently by myself and that I only used the references and auxiliary means indicated in this work.

Dresden, September 12, 2011

Maximilian Thilo Walther
Acknowledgements

This dissertation is the result of my research efforts in the context of the Aletheia project. Aletheia was a BMBF project with participation of the Chair of Computer Networks, Institute of Systems Architecture at the Faculty of Computer Science of Technische Universität Dresden. Other project members were ABB, BMW Group, DPDHL, Eurolog, Fraunhofer IIS, Freie Universität Berlin, Giesecke & Devrient, Humboldt-Universität zu Berlin, ontoprise, Otto GmbH & Co. KG, SAP AG, SAP Research, and TECO.

Though only my name appears on the cover of this dissertation, various other people have supported me during its production.

I want to express my gratitude to Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill for his constant encouragement. The biannual conventions in various places around Dresden as well as periodic feedback concerning my work helped me write my dissertation successfully. I would also like to thank my second adviser Prof. Dr. rer. nat. Claudia Linnhoff-Popien as well as my scientific consultant (Fachreferent) Prof. Dr.-Ing. Michael Schroeder for providing helpful feedback.

I am obliged to many of my colleagues who supported me during the research process, especially to Dr. Daniel Schuster who has already advised me during my studies of media and computer science. Various discussions with my Aletheia colleagues accounted for a better understanding of different sets of problems and possible solutions. I have had support from several students during the development of new ideas and components for Fedseeko. Finally, for completing the dissertation, many proposals for changes have been provided by other chair members. I want to thank all of you for each bit you have provided to me in order to successfully finish my work.

Last but not least I also want to show my gratitude to Babette, my parents, my brothers, Babette’s family, and my friends. You allowed me to keep up my motivation while also enjoying the sweets of life.

1Throughout the dissertation, the pronoun “she” (respectively, “her”) will be used when speaking of a generic person. This kind of linguistic usage is not meant to be offensive in any way and refers to human beings of both genders.
Abstract

Product information search has become one of the most important application areas of the Web. Especially considering pricey technical products, consumers tend to carry out intensive research activities previous to the actual acquisition for creating an all-embracing view on the product of interest. Federated search backed by ontology-based product information representation shows great promise for easing this research process.

The topic of this thesis is to develop a comprehensive technique for locating, extracting, and integrating information of arbitrary technical products in a widely unsupervised manner. The resulting homogeneous information sets allow a potential consumer to effectively compare technical products based on an appropriate federated product information system.
# Contents

1. **Introduction** 1  
   1.1. Online Product Information Research 2  
      1.1.1. Current Online Product Information Research 2  
      1.1.2. Aspired Online Product Information Research 3  
   1.2. Federated Shopping Portals 5  
   1.3. Research Questions 7  
   1.4. Approach and Theses 7  
      1.4.1. Approach 8  
      1.4.2. Theses 9  
      1.4.3. Requirements 10  
   1.5. Goals and Non-Goals 11  
      1.5.1. Goals 11  
      1.5.2. Non-Goals 11  
   1.6. Contributions 11  
   1.7. Structure 12  

2. **Federated Information Systems** 13  
   2.1. Information Access 16  
      2.1.1. Document Retrieval 16  
      2.1.2. Federated Search 19  
      2.1.3. Federated Ranking 25  
   2.2. Information Extraction 28  
      2.2.1. Information Extraction from Structured Sources 29  
      2.2.2. Information Extraction from Unstructured Sources 30  
      2.2.3. Information Extraction from Semi-structured Sources 31  
   2.3. Information Integration 46  
      2.3.1. Ontologies 46  
      2.3.2. Ontology Matching 51  
   2.4. Information Presentation 65  
   2.5. Product Information 65  
      2.5.1. Product Information Source Characteristics 65  
      2.5.2. Product Information Source Types 68  
      2.5.3. Product Information Integration Types 70  
      2.5.4. Product Information Types 71  
   2.6. Conclusions 73  

3. **A Federated Product Information System** 75  
   3.1. Finding Basic Product Information 76  
   3.2. Enriching Product Information 77  
   3.3. Administering Product Information 78  
   3.4. Displaying Product Information 78  
   3.5. Conclusions 79
## Contents

### 4. Product Information Extraction from the Web

4.1. Vendor Product Information Search ........................................... 82
   4.1.1. Vendor Product Information Ranking ..................................... 83
   4.1.2. Vendor Product Information Extraction .................................. 89

4.2. Producer Product Information Search ........................................... 94
   4.2.1. Producer Product Document Retrieval ..................................... 95
   4.2.2. Producer Product Information Extraction .................................. 99

4.3. Third-Party Product Information Search ...................................... 110
4.4. Conclusions ............................................................................. 111

### 5. Product Information Integration for the Web .................................. 113

5.1. Product Representation .......................................................... 115
   5.1.1. Domain Product Ontology ...................................................... 115
   5.1.2. Application Product Ontology ................................................. 117
   5.1.3. Product Ontology Management ............................................... 118

5.2. Product Categorization ............................................................. 119
5.3. Product Specifications Matching ................................................. 120
   5.3.1. General Procedure ............................................................... 120
   5.3.2. Elementary Matchers ............................................................. 122
   5.3.3. Evolutionary Matcher .............................................................. 128
   5.3.4. Naïve Bayes Matcher ............................................................... 131
   5.3.5. Result Selection ................................................................. 132

5.4. Product Specifications Normalization .......................................... 135
   5.4.1. Product Specifications Atomization ......................................... 135
   5.4.2. Product Specifications Value Normalization ............................. 135
5.5. Product Comparison ................................................................. 136
5.6. Conclusions ............................................................................. 136

### 6. Evaluation ............................................................................. 139

6.1. Implementation ........................................................................ 139
   6.1.1. Offers Service .................................................................. 141
   6.1.2. Products Service ................................................................. 143
   6.1.3. Snippets Service ................................................................. 149
   6.1.4. Fedseeko ........................................................................ 149
   6.1.5. Fedseeko Browser Plugin ...................................................... 152
   6.1.6. Fedseeko Mobile ................................................................. 154
   6.1.7. Lessons Learned ................................................................. 155

6.2. Evaluation ............................................................................. 157
   6.2.1. Evaluation Measures ............................................................ 157
   6.2.2. Gold Standard ................................................................ 159
   6.2.3. Product Document Retrieval ................................................. 162
   6.2.4. Product Specifications Extraction ...................................... 165
   6.2.5. Product Specifications Matching ........................................ 169
   6.2.6. Comparison with Competitors ............................................ 174
6.3. Conclusions ................................................................. 174

7. Conclusions and Future Work ........................................... 177
  7.1. Summary ................................................................. 177
  7.2. Conclusions ............................................................. 178
  7.3. Future Work ............................................................ 179

A. Pseudo Code and Extraction Properties ................................. 181
  A.1. Pseudo Code ............................................................ 181
  A.2. Extraction Algorithm Properties .................................... 182
    A.2.1. Clustering Properties .......................................... 182
    A.2.2. Purging Properties ............................................. 185
    A.2.3. Dropping Properties ........................................... 185

B. Fedseeko Screenshots ..................................................... 187
  B.1. Offer Search ......................................................... 187
  B.2. Product Comparison ................................................ 191
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Mockup for a Product Detail View in a Federated Search Portal.</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Product Information from Vendors, Producers, and Third Parties.</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Related Research Topics for Federated Information Search and Integration.</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Different Steps to be Taken in a Federated Information System.</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>A Document Retrieval System. [124]</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>The Federated Architecture of CrIP. [145]</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Architecture of the IPIS System. [110]</td>
<td>21</td>
</tr>
<tr>
<td>2.6</td>
<td>RDQL Example Query for the IPIS System.</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>The Federated Product Information Management Architecture of Aletheia.</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>Relationship of Extraction Quality and Source Structure in IE.</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>Amazon Product Advertising API Result in XML.</td>
<td>30</td>
</tr>
<tr>
<td>2.10</td>
<td>Template-based Web Page Generation.</td>
<td>31</td>
</tr>
<tr>
<td>2.11</td>
<td>HTML Code of an SLR Camera Page.</td>
<td>34</td>
</tr>
<tr>
<td>2.12</td>
<td>HTML Code of a Camcorder Page.</td>
<td>34</td>
</tr>
<tr>
<td>2.13</td>
<td>PAT Tree Representation for Extracts of HTML Code in Figure 2.12.</td>
<td>39</td>
</tr>
<tr>
<td>2.14</td>
<td>RoadRunner Example for the HTML Code in Figure 2.11 and 2.12.</td>
<td>40</td>
</tr>
<tr>
<td>2.15</td>
<td>Different Types of Schemas and Instances.</td>
<td>47</td>
</tr>
<tr>
<td>2.16</td>
<td>The Ontology Hierarchy.</td>
<td>49</td>
</tr>
<tr>
<td>2.17</td>
<td>Representation of Digi SLR 38 with GoodRelations and eClassOWL.</td>
<td>50</td>
</tr>
<tr>
<td>2.18</td>
<td>Typical Sequence of Tasks in a Schema Matching Process.</td>
<td>52</td>
</tr>
<tr>
<td>2.19</td>
<td>Schema-based Matching Techniques. [157]</td>
<td>53</td>
</tr>
<tr>
<td>2.20</td>
<td>Example Schemas from the Digital Camera Domain.</td>
<td>55</td>
</tr>
<tr>
<td>2.21</td>
<td>Similarity Flooding for two Minimal Schemas about Photo Cameras.</td>
<td>57</td>
</tr>
<tr>
<td>2.22</td>
<td>Example for a Decision Tree of MatchPlanner. (inspired by [50])</td>
<td>58</td>
</tr>
<tr>
<td>2.23</td>
<td>Characterization of the Amazon Website.</td>
<td>68</td>
</tr>
<tr>
<td>2.24</td>
<td>Cube Taxonomy for Product Information Search and Integration.</td>
<td>71</td>
</tr>
<tr>
<td>3.1</td>
<td>The FEAD Chain - Find, Enrich, Administrate, and Display Information.</td>
<td>75</td>
</tr>
<tr>
<td>3.2</td>
<td>Offer Information from Vendor Sources.</td>
<td>76</td>
</tr>
<tr>
<td>3.3</td>
<td>Product Information from Producer Sources.</td>
<td>77</td>
</tr>
<tr>
<td>3.4</td>
<td>Product Information in a Clean Format.</td>
<td>78</td>
</tr>
<tr>
<td>3.5</td>
<td>Presentation of Product Information.</td>
<td>79</td>
</tr>
<tr>
<td>4.1</td>
<td>The FEAD Chain - Find Basic Product Information.</td>
<td>81</td>
</tr>
<tr>
<td>4.2</td>
<td>Vendor Product Information Search.</td>
<td>82</td>
</tr>
<tr>
<td>4.3</td>
<td>Source Ranking Algorithm.</td>
<td>85</td>
</tr>
<tr>
<td>4.4</td>
<td>Product Ranking Algorithm.</td>
<td>88</td>
</tr>
<tr>
<td>4.5</td>
<td>Visual Example for an Information Extraction Wrapper.</td>
<td>89</td>
</tr>
<tr>
<td>4.6</td>
<td>The FEAD Chain - Enrich Basic Product Information.</td>
<td>94</td>
</tr>
<tr>
<td>4.7</td>
<td>Product Specifications Page Retrieval Overview.</td>
<td>96</td>
</tr>
<tr>
<td>4.8</td>
<td>Scoring Potential Product Specifications Pages.</td>
<td>97</td>
</tr>
<tr>
<td>4.9</td>
<td>Extraction of Product Specifications from the Specifications Page.</td>
<td>101</td>
</tr>
</tbody>
</table>
List of Figures

4.10. Clustering of Product Web Page Content. 101
4.11. Example Representation of an HTML Element in XML. 102
4.12. Important XPath Queries for the Wrapper Configuration. 109

5.1. The FEAD Chain - Manage Product Information. 113
5.2. General Process for Product Information Integration. 114
5.3. Product Domain Ontology. 116
5.4. Excerpt of the Product Application Ontology with Instances. 117
5.5. Ontology Matching of Product Specifications. 121
5.6. Example Unit Model for the Distances Domain. 127

6.1. The Prototype’s Federated Product Information System Architecture. 140
6.2. Screenshot of the Offers Service. 141
6.3. Offers Service Classes. 142
6.4. Product Page Locating Classes. 144
6.5. Product Specifications Extraction Classes. 145
6.6. Product Specifications Matching Classes. 147
6.7. Screenshot of the Products Service. 148
6.8. The FEAD Chain - Present Product Information. 149
6.9. Screenshot of Fedseeko’s Offers Search Interface. 150
6.10. Screenshot of Fedseeko’s Product Comparison Interface. 151
6.11. Provocation of Denial-of-Service Attack Reactions by the Offers Service. 152
6.13. Screenshots of Fedseeko Mobile. 154
6.15. Screenshot of the Gold Standard Manager. 159
6.16. Distribution of Products over Categories. 161
6.17. Product Page Locating Effectiveness without Domain Knowledge. 162
6.18. Product Page Locating Effectiveness with Domain Knowledge. 163
6.20. Product Specifications Extraction Effectiveness. 166
6.23. Elementary Matchers Effectiveness. 170
6.24. Evolutionary Matcher Effectiveness. 171
6.25. Naïve Bayes Matcher Effectiveness. 172
6.26. Product Category Coverage by Google Products. 175

B.1. Query Suggestions in the Offer Search View. 187
B.2. Cross-Site Request Plugin Interaction during Offers Page Retrieval. 188
B.3. Offers from Amazon, Ebay, and Evendi.com for “easyshare”. 188
B.5. Detail View including Preview on (unmatched) Product Specifications for Kodak Easyshare C195 Digital Camera (Purple). 190
B.6. Settings Window of Cross-Site Request Plugin with Cursor Pointing on
Previously Used Option. ........................................ 191
B.7. Overview of Available Categories for Product Comparisons. .......... 191
B.8. Available Facets for Digital Camera Category with two Breadcrumbs. . 192
B.9. Comparison of Remaining Products for Chosen Facets. ................. 193
List of Tables

1.1. Criteria for a Comprehensive and Balanced Product Information Source. . 8
2.1. Comparison of Architectures for Federated Information Search. . . . . . 24
2.2. Comparison of Approaches for Federated Ranking. . . . . . . . . . . . . 27
2.3. Comparison of Approaches for IE from Semi-structured Sources. . . . . 45
2.4. Comparison of Approaches for Schema and Ontology Matching. . . . . . 63
2.5. Source Characteristics of Vendors, Producers, and Third Parties. . . . . 70
4.1. Requirements for a Federated Ranking Algorithm. . . . . . . . . . . . . 84
4.2. Example Scores Calculated by the Ranking Algorithms. . . . . . . . . . 99
5.1. Thresholds and Weights of the Composite Matcher. . . . . . . . . . . . . 129
5.2. Example Matrix created by the Composite Matcher. . . . . . . . . . . . . 132
5.3. Example Matrix after the Selection Step. . . . . . . . . . . . . . . . . . . 133
6.1. Contents of the Created Gold Standard. . . . . . . . . . . . . . . . . . . 160
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABox</td>
<td>Assertion Box</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AVP</td>
<td>Attribute Value Pattern</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DDoS</td>
<td>Distributed Denial-of-Service</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial-of-Service</td>
</tr>
<tr>
<td>DR</td>
<td>Document Retrieval</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>FEAD</td>
<td>Find, Enrich, Administrate, and Display</td>
</tr>
<tr>
<td>FN</td>
<td>False Negative</td>
</tr>
<tr>
<td>FP</td>
<td>False Positive</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IE</td>
<td>Information Extraction</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IR</td>
<td>Information Retrieval</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JSSH</td>
<td>Java Secure Shell</td>
</tr>
<tr>
<td>LFEQ</td>
<td>Large and Frequent Equivalence Class</td>
</tr>
<tr>
<td>LGG</td>
<td>Least General Generalization</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>OAEI</td>
<td>Ontology Alignment Evaluation Initiative</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>ODT</td>
<td>Open Document Format</td>
</tr>
<tr>
<td>OEP</td>
<td>Object Extraction Pattern</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PDF</td>
<td>Portable Document Format</td>
</tr>
<tr>
<td>PIM</td>
<td>Product Information Management</td>
</tr>
<tr>
<td>PMI</td>
<td>Pointwise Mutual Information</td>
</tr>
<tr>
<td>POS</td>
<td>Part of Speech</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>RDQL</td>
<td>RDF Data Query Language</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SLR</td>
<td>Single-Lens Reflex</td>
</tr>
<tr>
<td>SOAP</td>
<td>originally: Simple Object Access Protocol</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol and RDF Query Language</td>
</tr>
<tr>
<td>SPARUL</td>
<td>SPARUL Protocol and RDF Update Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TBox</td>
<td>Terminology Box</td>
</tr>
<tr>
<td>TF-IDF</td>
<td>Term Frequency - Inverse Document Frequency</td>
</tr>
<tr>
<td>TP</td>
<td>True Positive</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery, and Integration</td>
</tr>
<tr>
<td>UFRE</td>
<td>Union-Free Regular Expression</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UNSPC</td>
<td>United Nations Standard Products and Services Code</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformation</td>
</tr>
</tbody>
</table>
With its introduction by Tim Berners-Lee in 1989 [20] the World Wide Web (WWW) revolutionized the way people use the Internet. Its original user group, mainly consisting of researchers in the computer science area using it for scientific information exchange, altered dramatically as the WWW suddenly allowed straightforward information publishing for an arbitrary amount of people by comfortable means. The Web at that time is called “Web 1.0” in retrospect. As Internet usage times increased, people dislocated miscellaneous activities onto the Web, including communication, file exchange, research, gaming, and e-commerce. Concerning the e-commerce section, an enormous amount of online malls emerged offering a large variety of products. Producers reacted to the arising possibilities by presenting their products in an appealing and informative way online. Thus, the WWW has become the modern day’s most important shopping facility.

Consumers have availed themselves of the given possibilities by carrying out a major part of their purchases online. 20 million Germans were already using online shopping sites to make purchases as early as 2003 [72]. By 2005 about 25.2 million Germans were using online shops [73], that is, about the half of 14- to 69-year-olds were benefiting from the Web’s latest development.

By introducing more and more web applications and tools for layman information publishing, the Web ascended to a superior level, generally going by the name of “Web 2.0”. Today, any average Internet user may publish arbitrary information, thus changing the Web to an unmanageable data assemblage comprising every kind of imaginable content. Consequently, e-commerce has experienced another boost as the Web 2.0 now enables people to discuss and rate products. In this way the Web not only allows users to buy products but also allows them to form opinions about products from the massive quantity of user-generated content at hand.

This development has also lead formerly small online shopper groups like female consumers and senior citizens to increasingly discover the Web as a shopping platform [75]. Between 2003 and 2007 the number of online shoppers increased by 41% (from 20.2 million to 28.6 million people in Germany) with a growth rate of 50% for the female consumer group and 79% for people older than 50 years. The development from a provider-centric Web to a user-centric Web is also reflected in different online shopping surveys. For example, it has been proven that vendor comparisons are getting more important to online shoppers [76] since, in 2007, approximately 44% of Germans informed themselves about electronic products online before purchasing while in the domain of
flight tickets as many as 70% of them searched for the cheapest offer online. The most popular online malls in 2008 by far were eBay [51] (26%) and Amazon [98] (22%) [74]. Since 2009 Amazon is the most popular online mall in Germany with about 16.7 million buyers per year.

Nielsen’s consumer report on trends in online shopping [26] states that in 2008 about 86% of people worldwide with an Internet connection (with a peak of 99% in Korea) had already made purchases over the Internet. The most popular product category was that of books (41%), followed by clothing, airline tickets, and various electronic articles. Compared to 2005, clothing had the highest growth rate in online shopping. The majority of people used credit cards for payment (60%) and returned to sites they had already visited for previous purchases.

These figures confirm that the popularity of online shopping and online product information collection are more and more eliminating the traditional way of shopping. Unfortunately, moving the process of information collection onto the Internet results in the absence of client counseling, which forces consumers to gather product information on their own. Finding and consolidating this information is an ambiguous challenge as it is distributed all over the Web, thus obliging the consumer to locate and evaluate sources, extract relevant information, and integrate it. Additional problems emerge if a user does not prefer a special product in advance or is even missing basic knowledge about the product’s domain.

The following scenarios describe the current problems in online product information research as well as the potential for optimizing the whole process. This optimization would mean another boost for the online shopping sector.

1.1. Online Product Information Research

In the following, two scenarios, located in the area of online product information research, will be described. Firstly, a characteristic sequence of steps to be taken before actually buying a product is outlined that is typical for the present situation. Then, a desired scenario is described that shows in which manner product information research could be simplified.

1.1.1. Current Online Product Information Research

A consumer with average knowledge of the digital camera domain is willing to buy a new digital camera. She does not prefer any special producer and is only interested in the specifications of the camera and other users’ opinions. Thus, she uses her browser to navigate to a search engine, e.g., Google [100], and enters the query “digital camera”. A list of some million web pages is retrieved from Google’s index and ordered using its extended PageRank algorithm [136]. The consumer examines the search results and may also find links to so-called federated shopping portals, which will be described in more detail later on. She finally navigates to a test web page presenting different cameras including expert and user opinions. The prospective buyer inspects some of them to learn about the different brands on the market as well as camera models produced by them.
Going through the detailed descriptions of the cameras, she gets an idea of which features would be of interest for her. For example, she identifies a desired camera resolution between 10 and 12 Megapixels and a maximum aperture between three and four. Finally, she takes out a piece of paper and writes down some models which seem suitable.

Then, the consumer wants to know more about the camera models she has selected and does some specific searches on Google for each of the selected models. She finds the websites of corresponding producers which provide more details on the cameras. Additional test web pages are located by the consumer that present information on the cameras of choice. Each web page is inspected by the consumer and conflicting statements concerning user opinions and product specifications are merged manually. Valuable information is added to the piece of paper to get a clearer picture of the potential camera to be bought.

Finally, the consumer chooses one of the examined cameras. She goes to an online shop, such as Amazon, enters a query for the camera of choice and inspects it again. Different versions of the camera are available on Amazon. Thus, the user examines all of them and finally selects the camera she wants to buy. The last step in the activity chain consists of going through the purchasing process and ordering the camera.

As shown in the scenario, the product information collection process to be executed by the consumer consists of a series of steps involving several information sources that are unknown in advance. The scenario does not assume that the consumer is an expert in the product category she is doing her research in. Still, comprehensive knowledge of online research is required to execute the different steps as a lot of problems may emerge during the whole process.

Such problems might include badly-ranked web search result lists, e.g., if the search engine misinterprets given queries. They also include test web pages with low content quality, thus creating an incorrect image of the product category which is of interest to the user. Additionally, this way of manually collecting product information produces incomplete product and product category views. It also requires some work to locate specific product web pages, such as a product’s web page on its producer’s domain, and of course to merge conflicting statements about the products. Using a well-known online shop for purchasing the actual product might also result in consumer dissatisfaction as the web shop might not offer the desired product or might not be the cheapest address at which to buy the product.

As the scenario illustrates, there is a lot of potential for easing this process. Before outlining the ideas to be developed in subsequent chapters, another scenario shows the aspired manner of online product information research.

1.1.2. Aspired Online Product Information Research

In contrast to the scenario described in the previous section, this scenario shows the sequence of steps to be taken for reaching the same goal assuming that a web application called Fedseeko, which implements a series of extraction and matching algorithms, is available. It therefore shows the necessity of easing the current product information research process.
A consumer with average knowledge of the digital camera domain is willing to buy a new digital camera. As before, she does not prefer any special manufacturer and is only interested in the specifications of the camera and other user opinions. Thus, she uses her browser to navigate to the Fedseeko website, chooses a list of interesting product vendors and enters a search query “digital camera”. Different products are retrieved from the queried online malls and presented in a reasonable order. The consumer inspects some products of interest and reviews their detail views.

A detail view consists of product information gained from different information sources. Online malls deliver a basic information set consisting of the product name, the producer name, a picture, and a price. Producer pages offer detailed product information that mostly includes images and product specifications, as well as descriptive texts. Third parties like boards and blogs offer all kinds of user-generated content especially including opinions on these products. An example of such a view is given in Figure 1.1. Different tabs are available for browsing through the available information snippets. Additional third-party information (products reviews, web search results, etc.) can be retrieved by clicking on the plus sign.

![Mockup for a Product Detail View in a Federated Search Portal.](image-url)
After having formed an opinion about what product characteristics are most important for the buying decision, the user decides to specify some characteristics in more detail, such as a desired camera resolution between 10 and 12 Megapixels and a maximum aperture value between three and four. This is done through a faceted search interface being offered by Fedseeko. The user enters the additional information and a result list is returned, having been adapted to the given details. Finally, she considers two cameras that fit her needs and thus asks Fedseeko to compare their specifications. As camera one is cheaper, the user finally decides to buy this camera and follows one of several links to navigate to any online mall on the Web.

When comparing both given scenarios, the advantages of a federated product search portal like Fedseeko become clear. The consumer uses a straight-forward process to first create an overview of the desired product category and then make a determined decision about which product to buy. This way, the final product decision becomes repeatable, that is, if the user would start to do her research under the same initial conditions (including her initial product domain knowledge, available product portfolio, etc.), she would end up with the same buying decision. Compared to the aspired manner of research, the process of information collection in the previous scenario is rather random. Depending on different factors a user might not find a particular information source that would have affected her buying decision and thus could easily choose a product that does not perfectly fit her needs.

The idea of gathering products from different online malls in one central repository is not new and thus so-called federated shopping portals have been developed that enjoy great popularity on the Web. Typically, such portals are only able to compare product prices. Some of them have started to gather detailed product information as well, hence, they are presented in the following section.

### 1.2. Federated Shopping Portals

Federated shopping portals such as Ciao! [78], Yatego [79], or Shopping.com [156] partly offer a solution to the problems described above. A series of such portals have emerged which try to allocate product information originating from various websites, especially online malls like Amazon, in one exclusive site. Their main goal is a federated collection of product offers which give the consumer the possibility to choose the cheapest shop at which to buy her product of interest. In rare cases, consumers are also informed about product details (i.e., specification data consisting of key-value pairs like “effective pixels: 9 MP” for a digital camera).

When including information from online malls, these systems are generally able to query available Web Services directly, providing product information in a structured manner, or get the offers by feed-like mechanisms. That is, they receive product updates through a limited Web Service like the one provided through Buy.com [25]. Including actual product specifications requires more manual work as the employees of the according information system company have to locate the producer’s website, find the web page presenting the product of interest, pinpoint the product information, and extract it.
Alternatively, consumers who are keen to share their knowledge with others can provide this information.

As this process evidently requires a lot of man hours, information providers tend to either specialize in concrete product domains (e.g., Digital Photography Review [10] on digital cameras) or reduce the presented information to very general details that all products have in common, such as a product name, a producer name, a picture, prices, etc. Eventually, user-generated content is also of interest, especially reviews on bought products and ratings concerning the shop having sold the corresponding product. This information is provided directly to the different portals by the consumers.

The information to be gathered is therefore located in three different classes of information sources, namely vendors, producers, and third parties. Figure 1.2 provides a general idea about these source types.

Vendors include all available online malls and thus can be numerous per product. Producers have assembled the product to be sold and are generally unique for each product. Third parties are sources not belonging to the first two categories and provide information generated by consumers or product testers.

As mentioned above, a lot of manual work is required to gather product information from the different source types. Some vendor sources might be included in a federated shopping portal using Web Services or technologies like the shopinfo.xml standard [185] that is based on a shopinfo.xml file describing the structure of a shop’s product information.
Vendors that do not offer their product catalogs in a structured machine-readable way cannot be included this easily. Web scraping technologies, i.e., information extraction on the Web, provide a means to generate structured information from semi-structured sources and thus offer a solution for this problem. Producers also have to be accessed using web scraping. In contrast to vendor sources, the amount of producer sources is unclear and cannot be known in advance as it is dependent on the product catalogs of included vendors. Hence, additional problems emerge as the producer site for each product has to be located on the Web when the corresponding product’s information is queried. Concerning third parties, some sources might be known in advance, e.g., product forums or test web pages, and thus can be analyzed generically before collecting a product’s information. Other third-party sources might only be located and analyzed at query time, thus showing the same problems as producer sources. These aspects lead to the central research questions presented in the next section.

1.3. Research Questions

Considering the class of problems mentioned above, the central research questions reduce to the following: How can basic product information be located and extracted as the bootstrapping information for follow-up steps? Which steps have to be taken to enhance this basic information with valuable product information from the Web, that is, how can additional information be found and extracted? In what way does a federated system need to organize such information to enable the comparison of products?

The goal of the following chapters is to answer these questions and to develop techniques for automatically locating and accessing all three product information source types while only requiring minimal user interaction. The main focus lies in the collection and unification of product specifications from producer sources, as this information enables effective product comparisons. Naturally, for developing the required algorithms, a series of related technologies are at the researcher’s disposal which are to be presented in chapter 2.

Having the research questions in mind, the following section consequentially proposes an approach for solving these problems and sets up theses to be proven throughout the evaluation phase.

1.4. Approach and Theses

This section is dedicated to the theses which compromise the basis of this work. They describe the main aspects to be proven concerning federated product information search and semantic product comparisons on the Web. Since their orientation is rather technical, in the following, a short section will present the basic approach of this work and introduce briefly some of the most important technologies.
1.4.1. Approach

As the scenario from section 1.1.2 suggests, the desired product view is a combination of product information gained from all three product information source types, namely, vendors, producers, and third parties. All such sources possess both assets and drawbacks in the consideration of information quality. For instance, producer websites provide correct, fresh, and verifiable information, but use advertising text for promotion purposes. Encyclopedias like Wikipedia belong to third parties and contain goal-oriented and fresh information, but are not immune to biased product characterizations. Fedseeko tries to fulfill all conditions required to be called a comprehensive and balanced information source.

Peralta [167] regards data freshness and data accuracy to be the most relevant information source criteria. When focusing on product information, an extended criteria list, presented in Table 1.1, must be fulfilled. The specified criteria have been compiled in particular for federated product information search but may be important for information sources in general, too. They will be used in the evaluation phase to review the developed prototype.

Table 1.1.: Criteria for a Comprehensive and Balanced Product Information Source.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completeness</td>
<td>All available information is included.</td>
</tr>
<tr>
<td>Correctness</td>
<td>All included information is correct.</td>
</tr>
<tr>
<td>Freshness</td>
<td>All included information is up-to-date.</td>
</tr>
<tr>
<td>Neutrality</td>
<td>The information is not biased.</td>
</tr>
<tr>
<td>Goal Orientation</td>
<td>All included information is relevant.</td>
</tr>
<tr>
<td>Comparability</td>
<td>The information follows a distinct schema.</td>
</tr>
</tbody>
</table>

To meet the various conditions described in the table, relevant information from all three sources has to be located, extracted, and integrated. These steps depend on the appropriate application of different technologies for every information source type.

Locating information requires information retrieval technologies as well as information extraction mechanisms, that is, information sources must be discovered using search engines and web crawlers while inspecting such sources for their suitability must be done via an examination of specific parts of the retrieved sources.

The extraction of domain-relevant information from previously identified sources depends on technologies, such as wrapper generation, to access web sources in a structured way. Different technologies are available for the respective source structures.

Finally, the information integration step requires the management of a target data model and the adoption of matching and mapping technologies integrating extracted information with that model. For example, a data model can be represented by an
ontology, which is a serializable and extendable way to manage knowledge from arbitrary domains. Ontologies make up the basis of the Semantic Web.

The presented approach can be summarized by the theses in the following section.

1.4.2. Theses

According to the described approach, a main thesis and four sub-theses can be set up and then proven in the subsequent research work.

**Product View Creation.** Using information extraction and semantic technologies it is possible to create an all-embracing view for nearly any technical product, including basic details, offer information, technical specifications, and, optionally, user-generated content, while only having information from the public Web at hand.

1. **Product Specifications Locating.** Product specification pages provided by corresponding producers can be located on the Web using specifically adapted unsupervised algorithms. This thesis can be proven by the prototype through comparing the located web pages with those of a manually created gold standard of producer pages.

2. **Product Specifications Extraction.** Independent of different layouts, product specifications can be extracted from public producer pages in an unsupervised manner. This thesis can be proven by the prototype through comparing its extraction results with a manually created gold standard containing adequate information.

3. **Product Specifications Integration.** Product specifications from different producers can be harmonized using a comprehensive product ontology, thus allowing product comparisons. This thesis can be proven by the prototype through comparing its automatic mapping results with those of a gold standard created by a human being.

4. **Product Comparison.** The resulting harmonized product view allows consumers to compare products faster and more effectively than by using current state-of-the-art methods. This thesis can be proven by developing a platform which presents the collected product information in a superior manner.

As already mentioned, the current work strongly focuses on gathering and mapping product information from producer sources. Thus, the theses are directed at this area. The extraction and integration of product information from vendor sources, as well as ranking such information, is not the original focus of this work. Still, it must also be examined as well since some kind of bootstrapping information is needed for the follow-up algorithms. Third-party information completes the desired product view and, thus, is relevant to a consumer’s product information research process. It will only be sparsely covered.

When breaking down the provided theses, a set of requirements can be postulated. These requirements are presented in the following section.
1.4.3. Requirements

The introduced theses describe the task to be accomplished on an abstract level. For designing adequate algorithms, a more detailed view of emerging requirements is needed. Hence, for each thesis, three requirements are given. They focus on the input data, general information which is available during the corresponding process, as well as the resulting output.

The first thesis discusses the topic of locating product specification pages. Corresponding requirements are as follows.

Req 1.1 The locating of producer pages should only be based on a product’s name as well as its producer’s name.

Req 1.2 The locating algorithm should also work without user-given hints on where to find the page.

Req 1.3 Even if different producer product web pages are available, the algorithm’s output should be the actual product specifications page.

The locating step is followed by the extraction of product specifications. Again, three requirements are given.

Req 2.1 The extraction routine should be able to extract information even when supplied with only one product specifications page.

Req 2.2 When no knowledge from previous extractions is given, the algorithm should still be able to identify the extraction targets.

Req 2.3 Independent of the actual page template, the results of the extraction process should be a list of product specifications adhering to the producer’s terminology.

With extracted product specifications at hand, a matching task is to be executed that fulfills the following requirements.

Req 3.1 Product specification matching should only be based on a given set of product specifications as well as an appropriately modeled ontology.

Req 3.2 A limited set of domain knowledge, e.g., in the form of concept or property synonyms, should suffice to execute the matching.

Req 3.3 The matching result should consist of a set of 1-to-1 mappings with high similarity values.

Since the fourth thesis is the least important one, it will not be inspected in more detail here.

All requirements will be considered during the development of the necessary algorithms. Their fulfillment is to be proven in the implementation part of the evaluation chapter (section 6.1). Predicated on the resulting prototype, the high-level theses can finally be proven in the evaluation in section 6.2.
1.5. Goals and Non-Goals

As pointed out in the theses, a complete federated product information portal involves the adoption of many different techniques. As not all of them can be covered in detail, this section points out the goals and non-goals of the work.

1.5.1. Goals

The research task aims at enabling federated product information search using vendor sources and automatically extending the retrieved information with details from semi-structured and unstructured sources, especially producer sources. Thus, some basic ideas will be presented concerning the employment of Web Services or web scraping technologies for gathering basic product information sets from online malls. The integration of this information, that is locating, integration, and matching, will not be automatized completely as these mechanisms are just a necessary by-product. The main focus lies on the inclusion of semi-structured product information. Hence, mechanisms for automating the product specifications page retrieval, specifications extraction, and specifications matching must be developed. Third-party information will only be mentioned for the sake of completeness.

Additionally, users may experience a strong boost in the effectiveness of product information search if dynamic comparisons of products based on their features would be enabled. The central requirement for offering product comparisons is the use of a unique product terminology that may be modeled as an ontology.

1.5.2. Non-Goals

Unstructured information is generally out of scope as the application of Natural Language Processing techniques for processing such information would open up a whole new research field requiring even more extensive research. In the field of semi-structured information, the complete integration process is only presented for producer information. Vendor information search will require some user interaction. Third-party information is integrated on a very low level. This is just to show the feasibility of integrating such information.

1.6. Contributions

As described in the previous sections, several techniques are to be developed for enabling the creation of an all-embracing view on products of interest. These techniques can be divided by the different information source types and will be outlined briefly in the following.

Concerning vendors, algorithms for easily integrating their product catalogs in a federated consumer product information system either using the Web Services offered by those vendors or web scraping technologies will be presented. As the integration through Web Services is trivial, the main focus of this part will lie on the integration of vendors
using web scraping. The algorithms enable the comfortable creation of web scraping configurations specifically adapted to the website designs of different vendors. These configurations can be used by a wrapper for accessing the online mall in a structured way. Additionally, a ranking algorithm is introduced that enables the resorting of retrieved product search results based on the category of the given search query.

For collecting producer information, algorithms that locate product specification pages on producer sites and extract the specifications automatically will be offered. The information is mapped to a unique terminology which enables the comparison of products. The product terminology is represented using an ontology, especially adapted to the category of electronic products. As the producer information accumulation holds a high research potential, it will be in the main focus of all chapters.

1.7. Structure

This chapter presented a motivation and scenario for the federated product information search architecture to be laid out in the following. Contributions, theses and goals completed the introductory section.

In the basics chapter (chapter 2), the focus will lie on federated information systems since all techniques necessary to build a sound foundation of the concept are located in this area. This chapter is divided into research areas concerning the information access (document retrieval, federated search, and federated ranking), the information extraction (information extraction from structured, semi-structured, and unstructured sources), the information integration (ontologies and ontology matching), and the information presentation and representation.

Based on the the previously stated goals, the FEAD Chain is developed in chapter 3. It offers the preferred way to create a homogeneous product information base by combining and enhancing the techniques of the basics chapter.

Chapter 4 goes into the details of the document retrieval and information extraction process. Algorithms for gathering vendor product information as well as producer product information are developed. A ranking procedure, a categorization approach, and learning algorithms are included as well. The most important outcome of the presented algorithms are sets of technical product specifications in heterogeneous formats. Third-party information search is only mentioned shortly.

In the information integration chapter (chapter 5), the representation of product information through semantic representation formats is introduced. A comprehensive reusable data model is designed for this task. Then, concrete matching algorithms based on a set of elementary similarities are developed. Evolutionary and machine learning algorithms enable the optimal adjustment for the matching task.

Chapter 6 gives a brief overview of the implementation and then focuses on the evaluation of all major algorithms. Different views on the developed techniques are considered with the help of effectiveness and efficiency measures.

The final chapter (chapter 7) summarizes the results, provides some conclusions, and offers additional ideas for future work.
The present work is based on three major research areas, i.e., information retrieval, information integration, and semantic computing. The goal for this chapter is to identify relevant algorithms from each of these areas enabling the development of techniques for creating a valuable product information base using the Web. Such a combination of approaches is to be located in the field of vertical search [8]. A vertical is a special domain covered by the corresponding search system in detail, such as finance, images, or news. In vertical search, the information retrieval algorithms are object-oriented and not document-oriented. For the present use case, the objects are products and all available machine-processable sources are inquired for information to enhance the view on such objects. An overview of the mentioned research topics touched by vertical search in the product domain is given in Figure 2.1.

It is to be seen that the main concepts of information retrieval adopted in this work are document retrieval and information extraction. Mechanisms for locating and extracting product information will be used heavily in the design and implementation part. Federated search builds the bridge to the information integration area since it is already concerned with gathering and ranking information from different sources. For this work, various vendor, producer, and third-party sources need to be integrated with each other. The main part of information integration consists of concepts like schema matching, especially ontology matching. When adopting such mechanisms in the product domain, the management of product specifications might experience a strong boost since products become comparable through the creation of a consistent view. Finally, semantic computing is of importance, too, since product information needs to be managed semantically, e.g., by the use of ontologies.

The concepts of federated information systems are not new to the computer science area. They have been adopted especially by database systems researchers embraced by the term unstructured information management. Typically, the most important steps to be performed by unstructured information management systems include information access, information extraction, information integration, information aggregation, and information presentation. Generally, the ideas of federated information systems can be translated to handle web information as well. However, the effective adaptation of each mentioned step requires significant developments especially tailored for web information management.

Chang et al. [30] describe web information integration systems by the concept of a
wrapper. Following their explanations, a wrapper is a program that wraps an information source to provide information in a structured way. The wrapping procedure includes querying sources like web servers, collecting resulting pages via HTTP, performing information extraction, and finally integrating this information with other data sources. As already mentioned by Chang et al., the task of information extraction has received the biggest attention in this research area which shifted the meaning of the term wrapper to only denote the extraction part. In the context of this work, a wrapper will be equal to an extractor, thus reviving the term of a federated information system. The term Web will be left out in the concept name since such information systems may likewise access local databases or company file shares.

**Federated Information System.** A federated information system is a system that accesses information from different sources using various mechanisms and protocols, extracts valuable information from those sources’ contents, integrates the gathered information by the use of different matching algorithms, and presents it in a reasonable way.
As described by the definition, all steps including access, extraction, and integration of information are performed on the Web as well. The aggregation of information resides in the data warehousing area and is thus of less importance in this case. Figure 2.2 provides a schematic overview of the mentioned steps.

![Diagram of Federated Information System Steps](image)

Figure 2.2.: Different Steps to be Taken in a Federated Information System.

All steps displayed in the figure touch some of the research areas pictured above. Hence, the following sections iterate through each of them and introduce research topics offering mechanisms to execute corresponding tasks.

Originally, the information access block includes concepts from the computer networks and the database systems area. Technologies from both areas are to be employed throughout the subsequent chapters. However, the actual focus of this work is not on interfaces for accessing database management systems or the Internet Protocol Suite
[23]. Instead, advanced topics based on these technologies are of interest. They include document retrieval techniques for locating product information sources like producer pages and federated search concepts. Federated search is all about dynamically or statically integrating different information sources into one central system. It will be used to design the architectural framework of the prototype. Federated ranking algorithms are analyzed as well since only the most valuable product information is to be utilized for creating a product information base. Federated ranking is challenging because rather few indicators for creating a reasonable ranking functionality are given in many cases. All brought up topics are covered by section 2.1.

Information extraction (section 2.2) is the major research area this work is located in. Since especially the product information domain is of importance for the following chapters, the presented information extraction approaches need to be rated concerning their suitability for this domain. In addition, the most important product information sources for this work are web pages, thus putting a special focus on the extraction from semi-structured sources.

Furthermore, concepts of information integration will be examined for being able to integrate extracted product information snippets with each other (section 2.3). As ontologies are to be employed for managing product information, ontology matching approaches will also be part of this section.

Information presentation is not investigated. Section 2.4 is only included since the theses presented above claim to improve consumer product comparisons with the technologies used in this work.

As all explanations in the following chapters heavily depend on a clear product domain terminology, section 2.5 will be dedicated to a series of definitions for the product information domain and the examination of important product information sources. The results of this examination finally build a clean base to design customized algorithms for locating, extracting, and integrating product information.

2.1. Information Access

As described above, the related work presented in this chapter will focus on concepts located on a higher information access level while technologies related to database access and file transfer are assumed to be given. Document retrieval is the most basic mechanism to satisfy users’ information needs. Concepts of document retrieval and systems being based on these concepts will be used to enable the access to product information, e.g., by locating producer websites offering product specifications. Federated search and ranking often build upon systems that adopt document retrieval technologies and integrate results from such sources with further concepts. Thus, document retrieval opens up this section followed by federated search and ranking especially focusing on product information.

2.1.1. Document Retrieval

Document retrieval (DR) is frequently referred to as information retrieval (IR) in literature. Since the term document retrieval better describes the actual task to be executed, it
will be regarded as a branch of information retrieval throughout this work. Manning’s “Introduction to Information Retrieval” [124] offers the following definition for document retrieval (the term “information” has been replaced by “document”).

**Document Retrieval.** Document retrieval is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers). [124]

The definition shows that document retrieval is not about informing a user on the subject of her inquiry. Rather, it informs her about the existence or non-existence and whereabouts of documents relevant for her request [114]. Thus, additional steps have to be taken to actually filter out all information being relevant for the request as to be described in a later section about information extraction. Another problem is the potential inequality of the user’s information need and the actual query sent to the document retrieval system. The information need is the topic about which the user desires to know more while the query is the user’s attempt to express this desire in a string representation to be processed by the DR system.

A DR system can be optimized for retrieving the best-fitting result list to a given search query using some kind of ranking algorithm. Having only a query string at hand, it cannot be optimized for responding to a user’s information need. So-called context information is needed to retrieve better results in this case. Having marked out the document retrieval domain, the general architecture of a document retrieval system is to be described.

**Document Retrieval System**

A document retrieval system consists of a number of components as pictured in Figure 2.3. In an ongoing task, a crawler recovers documents, e.g., on the Web. Each document is processed by a parser using different parsing linguistics. These might include techniques like tokenization, language identification, compound-splitting, stop word-removal, case-folding, stemming, etc. Tokenization splits up continuous text strings into character sequences that are grouped together as a useful semantic unit for processing. Language identification is particularly useful for the tokenization step as different rules need to be adopted depending on the language of the document. If the found language includes compounds (e.g., German that combines singular words to form a new word), compound-splitting might help to match parts of words with terms in the vocabulary. Stop word-removal skips terms with high appearance frequencies like “is”, “and”, or “by”. Case-folding reduces all letters to lower case, thus shrinking the retrieval system’s vocabulary significantly. Finally, stemming reduces words, especially verbs, to a standard form. This enables the matching of verbs being used in different conjugations. A more intelligent alternative for stemming is called lemmatization.

Possibly, the documents are saved in a cache. Their parsed contents are written into different indices. Each index has a special task. For example, the positional index enables the retrieval of documents with queries like “bought a new camera” where the exact order of query strings is relevant.
After having created indices of sufficient size, a user query can be sent to the retrieval system. First of all, the query is parsed using the same linguistics as for the documents. This is important, e.g., as the stemming function may alter the document contents heavily and thus also has to alter the query strings in the same way. The parsed query is evaluated on the indices. Additionally, a spelling correction algorithm is executed and its result is evaluated over the indices to offer the user alternative results in the case of a spelling error.

Finally, by using a ranking function that heavily depends on the adopted data model, a result page is created that provides a ranked list of calculated hits to the user. Available data models can be divided into set-theoretic, algebraic, and probabilistic models. Set-theoretic models represent documents as simple sets. The Standard Boolean Model is an example for such a model and classifies a document as relevant if it does or does not contain certain strings provided by the user. Algebraic models represent queries and documents as vectors. Each term of a query or document is a dimension in such a vector with its term weight as coordinate. Retrieved documents are ranked by comparing the angle between query and document vector. The vector-space model is the basic model in this category. Probabilistic models are the third class of information retrieval models. Having a retrieved search result set at hand, a probabilistic algorithm calculates the probability that a document is relevant under the condition that it was retrieved for the current query. For example, the Binary Independence Model represents a query or document as a vector with binary coordinates, each of which is one if the term is present and zero if it is not. Further information on information retrieval and data models can be found in [124].

Document retrieval technologies are heavily used on the Web and build the basis of numerous information retrieval systems. A branch of the document retrieval domain is concerned with so-called focused or topical crawling. Focused Crawling deals with the directed identification of web sources to find information for a previously defined domain.
The concept of vertical search mentioned in the beginning of this chapter is heavily based on focused crawling strategies. For this work, the domain to focus on is the product information domain.

However, the federated product information system to be developed does not need to crawl the whole or parts of the Web and adopt indexing strategies for creating an information base. Rather, it aims at employing existing systems for specifically identifying valuable product information and aggregating this information for single products. Basic product information sets either provided by a user or an online mall are used to locate very specific product specification pages as well as similar pages being required by the information extraction process. The topic of focused crawling will thus not be covered here.

Document retrieval is the first step in making web information accessible. Having different document retrieval or deep web systems at hand, federated algorithms can be developed that direct this information to the integration level. Thus, existing approaches are examined in the next section.

2.1.2. Federated Search

Federated search comprises techniques for simultaneously searching over several sources and merging retrieved results into a single result set. As an exception, in some systems, retrieved information is not integrated in the same data structure. Instead, it is presented to a potential user in the form of several information snippets, a so-called Mashup. Federated search is an essential technique for this work since product information needs to be gathered from various sources on the Web. A meaningful definition of federated search is given by Péter Jacsó.

Federated Search. Federated searching consists of (1) transforming a query and broadcasting it to a group of disparate databases or other web resources, with the appropriate syntax, (2) merging the results collected from the databases, (3) presenting them in a succinct and unified format with minimal duplication, and (4) providing a means, performed either automatically or by the portal user, to sort the merged result set. [105]

This definition addresses the relevant dimensions of source integration in federated search systems, being query mapping, result merging, duplicate detection, and result sorting.

The Microsoft glossary states that “in a federated search, users receive results from multiple search and retrieval systems, for example, from other search engines, commercial information services, or internal databases. Federation is the blending of results from multiple, often non-compatible, search systems” [102].

The phrase “often non-compatible” points up the varying complexity of searching over multiple sources. The sources that federated search algorithms are working on are usually databases or web resources. Considering the inclusion of databases, the querying mechanisms are standardized through the use of SQL. In this case, it suffices for the federating algorithms to integrate different schemas with each other. This task alone is complex as the section below about ontology matching will point out. In many cases,
the matching of such schemas can only be realized with plenty of handcraft. Additional problems emerge when the queried sources vary on other levels as well, e.g., the accepted query types being simple strings in a first source and SPARQL \[142\] queries in a second one.

The most important characteristics of current federated search systems are pointed out in the description of CrIP \[145\] (Construction Information Platform), a pilot specification for a federated information portal in the construction domain. The authors constitute that state-of-the-art systems need to be (1) human-centered and easy-to-use, (2) adaptive and configurable, (3) supporting existing search practices, (4) open and scalable, (5) ontology-centered, and (6) Web-centered. Following these principles, they designed a system architecture consisting of a kernel, several comprehensive core services as well as additional external services. The system architecture is presented in Figure 2.4.

![Figure 2.4.: The Federated Architecture of CrIP. \[145\]](image)

The kernel includes manager components that are accessed through the actual CrIP Manager and the end-user portal via SOAP \[82\] calls. The manager components federate the system’s core services using a SOAP client as well as a UDDI registry. The accessed services include the ontology service for offering the system ontology to other services, the visualization service for adapting retrieved documents to the requesting device and seven knowledge services. From these knowledge services only the Knowledge Indexer Service has been implemented which uses the system’s ontology for semantically indexing documents in the available corpus. Concerning the information collection, CrIP is able to crawl the Web by the use of several so-called Web Spider Teams and to index found
documents using the Apache Lucene framework [44]. Furthermore, retrieved documents are classified by their relevance for the construction domain. At query time, CrIP can expand a user’s query semantically by specializing it, by adding synonyms, or by adding related concepts, thus allowing the retrieval of relevant documents even if some of those documents do not contain the original query. The returned documents are clustered into groups sharing the same semantics to keep the result set well-arranged.

Since CrIP only focuses on the construction area, it is to be categorized as a vertical federated search system. Considering this narrow domain, the authors were able to prove that retrieved documents are more relevant to the user’s needs than the ones retrieved by a general purpose engine such as Google, hence proving the feasibility of the vertical search concept.

In the field of federated product search, Shopbots [60] emerged already in the mid 90’s and were the first step towards integration of multiple vendors in a federated product search system using web scraping. Scraping vendor pages caused a number of problems because it is error-prone and delivers incomplete information. The IPIS system [111, 110] (Intelligent Product Information Search) tries to overcome these problems as it uses Web Service interfaces and ontology mapping as key technologies. Its architecture is visualized in Figure 2.5.

In the IPIS system, the user creates a semantic product query with the help of a so-called configuration. A configuration consists of the product’s category and some attributes that are filled out with appropriate values. For example, a user could search for a digital camera like the one presented in the previously shown product page. The configuration she creates for this camera is then translated into a RDQL query. RDQL (RDF Data Query Language) [153] is a query language for RDF specified by the W3C. For our example, the query might look like in Figure 2.6.

By the use of ontology mappings, the product category is translated into the category name used in each online mall, respectively. Then, chosen attributes are mapped to the shopping malls’ terminologies as well. Finally, the original query is translated for the
shops using the previously detected mappings. The retrieved results from the different
shopping malls can be mapped reversely to the IPIS format and compared with the
original configuration created from the user’s query. By calculating the distance between
attribute values of retrieved products and user-configured attribute values, a basic ranking
can be accomplished and the user receives appropriate results for her query.

As a main drawback, this approach relies on the assumption that each shopping mall
has web service interfaces and is able to process semantic queries. This assumption is
unrealistic since only very few shopping malls offer such services. Thus, the federated
system to be developed in this work tries to access shopping malls in a more generic way,
that is, while it employs Web Services where possible, it also offers fallback mechanisms to
extract valuable information through web scraping in the case no such service is available.

Another federated search architecture has been developed in the Aletheia
project [181]. Aletheia offers a means for semantically integrating product information from
structured and unstructured sources. The project focuses especially on product lifecycle
information since it was carried out in cooperation with different industrial partners such
as SAP, BMW, or ABB with the goal of improving the product information management
in each of those companies. The resulting architecture can be seen in Figure 2.7.

The central layer of Aletheia is the repository layer that consists of a semantic repository
(OntoBroker [134]), an uncertain repository (Sesame [63]), and a user context repository
(also Sesame). The Semantic Repository is responsible for managing certain (in the
sense of reliable) information provided in the form of an ontology. This assumption is
feasible since the ontology content is managed by experts using available import tools
or the Update Service. Furthermore, a reasoner runs on the available information base.
Additional facts contained in databases are lifted semantically to extend the facts base
in the Semantic Repository. The Uncertain Repository contains information that is not
acquired by hand but collected by machines. It is fed by the Extraction Service that
accesses crawled documents and uses the Semantic Repository’s ontology to semantically
index retrieved documents. Thus, it allows not only searching for strings, but for concepts
as well. This is a major advantage as it enables integrated search on the Semantic as
well as the Uncertain Repository. The extraction component is built on UIMA [66] and
Aperture [2] while persistence of created indices is assured by Lucene [44].

Different frontend services provide functionalities for creating rich user interfaces,
the most important of which is the Semantic Search and Navigation Service. It allows
searching in all repositories, faceted navigation, etc. Different frontends have been

<table>
<thead>
<tr>
<th>SELECT ?x</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
</tr>
<tr>
<td>(?x, rdf:type, fed:digital_camera)</td>
</tr>
<tr>
<td>(?x, fed:manufacturer, &quot;digi&quot;)</td>
</tr>
<tr>
<td>(?x, fed:total_pixels, 10750000)</td>
</tr>
<tr>
<td>USING fed FOR <a href="http://www.fedseeko.com#">http://www.fedseeko.com#</a></td>
</tr>
</tbody>
</table>

Figure 2.6.: RDQL Example Query for the IPIS System.
implemented including a web frontend as well as an Android [99] application for accessing the Aletheia Service Hub.

Since Aletheia was intended to support different usage scenarios in the first place (one for each participating company), its major strength is a high adaptivity. By only changing the Semantic Repository’s ontology as well as tied information providers, the system may be applied to a completely different domain. Consequently, Aletheia offers effective adaptation while retaining universality.

This work’s underlying research emerged in the context of Aletheia. Some analogies can be identified between both projects. During the development of Aletheia as well as the prototype to be presented later on, conceptual synergies have been used continuously. However, the architecture itself as well as the different components could not be reused for the prototype since Aletheia focuses on company-internal product information that is mostly managed by experts. Information in the context of Aletheia needs to be fully reliable as it influences large-scale decisions concerning the company’s product portfolio.
Thus, the focus of Aletheia is on lifting data to a semantic level while the present work focuses on retrieval, extraction, and integration of product information available on the Web. Furthermore, errors occurring in all phases of this process do not have a considerable influence on a company’s fortune.

A more lightweight approach is the shopinfo.xml standard [185]. Shop operators may define a shopinfo.xml that can be downloaded by any shopping portal easily, providing both RESTful Web Services [62] as well as downloading an XML product file for shop federation. This standard enables the distribution of shopping mall offers through Web Services and is generally a good approach. However, the decision about using this standard is on the shopping mall provider’s side and hence does not bring any advantage for this work. Mechanisms for interacting programmatically with web interfaces still need to be implemented for guaranteeing generic shopping mall integration in the federated system. A comparison of the presented approaches is given in the following.

Comparison of the Approaches

As can be seen in the previous section, enabling federated search is mainly a question of designing an adequate architecture. The characteristics introduced for the CrIP system help to rate such architectures concerning their appropriateness. In the following, Table 2.1 compares the most important architectures of the above section. Federated search is not the main focus of the work, the table has therefore been kept quite short.

Table 2.1.: Comparison of Architectures for Federated Information Search.

<table>
<thead>
<tr>
<th>Year</th>
<th>Author</th>
<th>System</th>
<th>Human-Centered</th>
<th>Adaptive</th>
<th>Common Search</th>
<th>Open</th>
<th>Ontology-Centered</th>
<th>Web-Centered</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>Kim et al.</td>
<td>IPIS</td>
<td>Yes</td>
<td>No</td>
<td>Partly</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2010</td>
<td>Rezgui et al.</td>
<td>CrIP</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2010</td>
<td>Wauer et al.</td>
<td>Aletheia</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

As proven by the table, Aletheia supports the most of the features demanded by the CrIP authors. Especially concerning adaptivity, it outperforms the other systems by far since they are strongly tied to their domains. Existing search practices are supported by all three systems while IPIS chooses a new way to execute queries and retrieve relevant results that might be unfamiliar to the user. None of the systems is open. This is either the case because they just never reached the production stage (IPIS and CrIP) or because they are designed to work in closed domains like single companies (Aletheia). Scalability was no focus for all three systems, thus, it is assumed to be moderate. All three systems are ontology- and Web-centered, supposably the two most important properties in the research area of federated information system architectures.
The task of this work is to retrieve, extract, and integrate product information from the Web. From the listed systems only CrIP really operates on the Web with a focus on the construction domain. Aletheia supports web contents at a very low rate. None of the systems implements techniques for extracting high-quality information from semi-structured or unstructured sources. IPIS relies on the availability of Web Services for gathering product information. Like most of the available federated product information systems, it only aims at integrating shopping malls in the federated product information platform. Unfortunately, information relevant for buying decisions is not restricted to multiple vendors, but also comprises information on producer pages as well as third-party information, additional data, knowledge, or services [89]. Hence, for being able to create the all-embracing product view postulated in chapter 1, a more generic way of integrating product information has to be found. Likewise, more sophisticated mechanisms for federated ranking are of importance and will be presented in the next section.

2.1.3. Federated Ranking

Ranking techniques are essential for every information system, including the one to be developed in this work, since users expect search results to be ordered by their relevance. Ranking in federated search systems differs heavily from standard approaches in centralized web search engines, mainly caused by the absence of a linking structure and missing rich information on single results. The last point is due to the following reasons. Firstly, bound sources do not expose their entire knowledge to the public. Secondly, requesting detailed information on single results during query time is too time-consuming. A good introduction on federated search including some existing federated search systems is given in [70].

Concerning the steps to be taken in federated search, federated ranking is part of step two, that is, merging the results from the queried information sources. The following definition describes the general procedure of federated ranking algorithms.

**Federated Ranking.** Federated ranking is the task of (1) rating diverse information sources, (2) reordering retrieved results, and (3) merging these result lists to create an overall ranked result list adhering to given ranking criteria. Each step is dependent on the given query.

Following the definition, federated ranking can be divided in source ranking and result ranking. Classical approaches like the PageRank [136] algorithm solely rank search results by the number of citations (out-links) from other web pages. Source ranking only happens implicitly as a website provider is ranked higher when all its web pages are assigned with a high ranking score.

In federated search approaches, sources have to be rated for the current query before processing their results as they might be of varying quality and it is generally not possible to manipulate their internal ranking algorithms. After having scored each provider, the retrieved result lists can be examined to judge each result list’s content concerning its relevance for the query. Finally, the results lists are merged into one list which is delivered to the querying entity.
One of several federated ranking approaches was presented by Si and Callan [159] for modeling the retrieval effectiveness of search engines in a federated search environment. The developed algorithm estimates the quality of such search engines through the creation of sample result lists. Each search engine’s sample result lists are put into a central database. This database is also evaluated by an effective centralized retrieval algorithm which rates the documents contained in this database. Then, by comparing the ranks that the search engines have given to the different results with the ones of the centralized algorithm, a profile for each search engine can be created. The profile affects the final rank being assigned to a search engine’s results at query time since the results’ ranking values are estimated only based on the source rank without inspecting the contents. For example, if the federated system would access Google (G), Yahoo [103] (Y), and Bing [38] (B) and they would have been assigned with normalized source ranking values of decreasing size, respectively, the first ten results returned for a search like “digital camera shr38” might look like 

\[ G_1 G_2 Y_1 G_3 B_1 Y_2 G_4 G_5 Y_3 B_2 \]

where the index denotes the position of the result in the original search engine’s result list.

The idea of rating sources corresponding to the quality of the results they return is generally a nice idea. However, it is not quite clear why a centralized retrieval algorithm should be able to determine this rating since it cannot be considered as the single source of truth. Furthermore, the quality of retrieved results might also be dependent on the executed query. Some search engines are specialized on certain domains and their results would be rated bad undeservedly when provided with a query from this domain. Thus, some optimizations have to be adopted to make this basic idea justifiable.

Paltoglou et al. [138] covered the problem of hybrid results merging. The authors point out that two kinds of methods for merging results from different information sources had been analyzed so long. On the one hand, approaches had been developed that are solely based on the sources’ underlying ranking mechanisms which leads to an insufficient ranking quality. The second type are methods based on the exhaustive evaluation of documents which cause high costs in computation time and traffic. As a solution they propose a model approach that only relies on partial source evaluation and strong estimation of results.

The Federated Search portal Science.gov [152] can be viewed as another model approach. The engine searches millions of documents from over 38 databases which are part of 14 U.S. agencies hosted on various servers. A single query may return thousands of hits, therefore, a decent ranking method is indispensable. Over the years, three different real-time ranking methods were designed for Science.gov: QuickRank, MetaRank, and DeepRank. QuickRank calculates a document score based on the number of occurrences of the search terms in the document by evaluating the document’s title and text snippets. MetaRank differs from QuickRank by assuming that documents within the sources’ databases supply an amount of meta-data about their contents that is then evaluated by the ranking method. Currently, DeepRank is the most thorough and exhaustive method and was introduced in 2007. It evaluates the full text of documents to produce the most accurate results.

The FedLemur project [12] presented by Avrahami et al. is a federated search engine specifically designed for providing information to the Fedstats.gov [61] website to compare...
the overall performance of a single-database solution with a federated approach. The engine is based on the Lemur toolkit [42] for language modeling and retrieval. Inter alia, the presented approach ranks integrated search engines with a word histogram that is created for each engine. The histograms are built by querying the search engines with random queries and analyzing the first returned documents. Queries for a search engine like Amazon.com would create histograms containing terms like “product” or “shopping cart” with quite high occurrence counts and other terms like “camera” with a lower count. Thus, specific domains for the search engines are described which can be used at query-time.

In the academic research the already mentioned shopinfo.xml standard [185] sorts search results by characteristics connected to an immediate purchase (price, shipping time, etc.) while not considering the relevance of each result for the given query. The ranking is only determined by underlying sources.

A short comparison of the presented approaches is given in the following.

Comparison of the Approaches

Federated ranking approaches can be categorized by the help of different properties. The most significant ones include the source type (search engines, online malls, etc.), applied ranking features (document content, result snippets, etc.), and the ranking granularity (source-level or result-level). An overview of a corresponding categorization for the previously introduced ranking approaches is given in Table 2.2

<table>
<thead>
<tr>
<th>Year</th>
<th>Author</th>
<th>System</th>
<th>Approach</th>
<th>Source Type</th>
<th>Ranking Features</th>
<th>Ranking Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Frierson</td>
<td>Science.gov</td>
<td>Federation of Science Portals</td>
<td>Science Portals</td>
<td>Document Content, Dates</td>
<td>Result Level</td>
</tr>
<tr>
<td>2005</td>
<td>Si and Callan</td>
<td>-</td>
<td>Retrieval Effectiveness Profiles</td>
<td>Search Engines</td>
<td>Search Engine Rating</td>
<td>Source Level</td>
</tr>
<tr>
<td>2006</td>
<td>Wolter et al.</td>
<td>shopinfo.xml Shop-side Federation</td>
<td>Online Malls</td>
<td>Product Attributes</td>
<td>Provider Level</td>
<td></td>
</tr>
</tbody>
</table>
All federated ranking approaches, including the ones presented in the previous section, have the same problems in common. The first class of problems is related to the queried information sources: It is generally unknown if these sources are of high or low quality and which ranking mechanisms are employed for ordering their results. The second class is related to the retrieved results. It includes the dilemma of intensive result evaluation leading to a bad performance and imprecise result evaluation allowing a very good performance. A general problem concerning result evaluation is that only sparse information is available at query time.

The situation is even worse concerning online malls since there are no documents available that can be queried at all. The ranking mechanisms employed in online malls sometimes just order retrieved products alphabetically which is hard to integrate with shops that order their products by popularity. Thus, the approaches presented in this section can only be used as inspirations for the federated ranking mechanisms to be developed in chapter 4.

After having located interesting documents or product information search results, the most important snippets contained in these results need to be extracted. Therefore, a series of mechanisms for executing this task will be presented in the next section.

2.2. Information Extraction

Information extraction covers all techniques for focused information collection from arbitrary information sources and represents the second step to be performed in a federated information system. It is often seen as a part of information retrieval. Information extraction could be defined as follows.

**Information Extraction.** Information extraction (IE) is the process of automatically gathering structured information for a defined domain or topic from structured, semi-structured, or unstructured machine-readable sources. IE adopts methods from various research areas including artificial intelligence and computer linguistics.

As stated in the definition, three differently structured information source formats are relevant for IE. The quality of extracted information is highly dependent on this structure since applied approaches make intensive use of it. As Figure 2.8 (inspired by Chang et al. [30]) shows, the information sources are harder to maintain by machines, the less structured the provided source is. Template-generated HTML is printed in bold letters since it will be in the focus of section 2.2.3.

Goal of the information extraction algorithm is always to enable the access of a source in a structured manner, regardless of the original source format. The following sections introduce the basic ideas for information extraction from all three types. The main focus is on information extraction from semi-structured sources, especially from template-generated HTML like marked in the figure, since product information on the Web is generally presented in a semi-structured format. Thus, this research area will be presented as the last and by far biggest section.
2.2.1. Information Extraction from Structured Sources

Structured sources are sources like databases or XML files with a determined schema. A database’s schema is given by the table descriptions while the XML schema may be determined by an XSD or DTD file. Information extraction from structured sources handles such source types. For both cases standardized APIs have been defined enabling the comfortable access to contained information. Such APIs will be adopted in this work since product information needs to be managed in a database and structured data formats like XML are generally used for providing product information through Web Services.

ODBC (Open Database Connectivity) [101] and JDBC (Java Database Connectivity) [39] are widespread access mechanisms for relational databases. The query language for accessing databases is typically some SQL dialect. XML can be accessed using a DOM [87] or SAX [125] parser. The DOM, or Document Object Model, is an interface for creating tree representations of XML files and thus enables easy access to elements and attributes in that tree. A SAX parser sequentially parses XML files and reacts by executing callback functions on registered events. Since Web Services mostly offer database contents as XML files, they are classified as structured sources, too. One such Web Service is the Amazon Product Advertising API [97]. An example for a made up camera returned by the Amazon service is presented in Figure 2.9.

As can be seen, the structure of the XML document is well-defined and thus predictable. When working with structured data, the accessing component generally knows both the schema and the location of the information. Retrieving the desired information is cheap since the requirement for a structured access of available information is already
satisfied. Thus, working on structured information is often not categorized as a type of information extraction. For the sake of completeness and since XML is often seen as a semi-structured source, it was still covered sparsely. The most challenging IE task deals with unstructured sources and is presented in the next section.

### 2.2.2. Information Extraction from Unstructured Sources

Unstructured information does not contain any structure except the grammar of the used language. Still, information extraction from continuous texts tries to make information from these sources available in a structured manner. The processing of unstructured information on the Web is gaining more and more interest since it enables the automatic processing of user-generated content. However, as can be seen in Figure 2.8, this type of information extraction generates results of very low quality. As the algorithms to be developed aim at creating a valuable product information base that allows effective product comparisons, unstructured sources will be out of scope. This section is thus kept
short intentionally.

The technology for processing unstructured information is called Natural Language Processing (NLP). NLP tries to detect entities like persons or organizations and relationships between those entities. Basic NLP techniques include tokenization, lemmatization, retrieving words from dictionaries for recognizing semantics, sentence structure parsing, etc. Tools like WordNet [127] can help a program to gain basic knowledge about semantic relations. Widely accepted libraries for NLP are GATE [43] (General Architecture for Text Engineering), LingPipe [14], and UIMA [66] (Unstructured Information Management Architecture).

The following section will focus on the most important type of information extraction for this work, that is, IE from semi-structured sources.

2.2.3. Information Extraction from Semi-structured Sources

Online product information being relevant to create an all-embracing product view is generally presented in a semi-structured manner. Thus, corresponding sources are notably important for this work. Information from semi-structured sources does not follow an explicitly determined schema. Instead, it may have an implicit schema which is usually not fully traceable since the information and the presentation part of the information source are mixed up. A representative format for semi-structured information, especially in the context of online information retrieval, is HTML. If talking about information extraction from HTML, generally the term web scraping is used. Web scraping is the process of extracting text contents from web pages over HTTP. The extracted text is then transformed into a given format, e.g., to be used on other websites. A legal web scraper has to respect the robots.txt file being located on every web server that states if the contents of a website may be processed by machines.

As declared in the introductory section, also information extraction from semi-structured sources tries to enable the access of contained information in a structured manner. This makes sense since semi-structured sources often consist of template-generated HTML, that is, database entries which have been inserted into a basic HTML page. This process is pictured in Figure 2.10.

---

![Figure 2.10.: Template-based Web Page Generation.](image-url)
Thus, the goal of information extraction in this case is the back transformation of contained information into its original format. In contrast to IE from structured sources, only few standard mechanisms are available to complete the task. Since, in the worst case, neither the structure of the source to be processed, nor the exact format of the information to retrieve is known, the IE task gets even more complex. Meta-information for describing the web page’s structure like XPath queries [18] or regular expressions [107] is helpful. Information extraction from semi-structured sources is an imprecise process. Thus, the hit rate and the quality of extracted information can be ameliorated by providing more such meta-information.

Several approaches dealing with IE from semi-structured sources have been developed in related research works. These approaches may be divided by several dimensions. The most important differentiation was proposed by Chang et al. [30, 29]. They classify IE tools by their degree of automation as manual, supervised, semi-supervised, or unsupervised approaches. Manual approaches rely on a programmer that creates wrappers specifically adapted to concrete extraction tasks. Such systems gain results of high quality but require a massive amount of user interaction. Supervised systems rely on a set of labeled training documents. Thus, the interaction with a user is required here as well. However, since the user is not working on the system’s program code, she does not need to be a programmer herself which is already a major improvement in comparison to manual approaches. Still, a lot of work is required for labeling the training documents before actually being able to extract desired information. Semi-supervised systems do not need such a labeled set of training documents. Instead, they try to extract valuable information directly and ask the user to mark the correct results afterwards. Using such user feedback, they are able to generate the correct wrappers and extract the desired information from similar pages automatically. In this case, the user interaction is minimized to boolean decisions concerning extraction results. Finally, unsupervised IE systems are approaches with the highest level of automation. Sources are analyzed by the corresponding algorithms and extracted information is provided directly to a user or a superior application. Naturally, such approaches gain worse results than approaches with user interaction.

Another important dimension for differentiating IE approaches in the semi-structured domain is the extraction target introduced by Sarawagi [148]. According to her, information extraction approaches can be subdivided into record-level, page-level, and site-level extraction systems. The first assume lists of similar data records available in special areas of a web page and try to extract these records by identifying boundaries between them. Page-level systems extract all data included in given pages. Finally, site-level approaches collect the information from all pages of a chosen website. Record-level systems can only identify data records which are presented through repetitive patterns. On the other hand, page- and site-level systems suffer from the problem of similar page retrieval. In most approaches this additional step is ignored although it is of high importance for the practical adoption of page-level information extraction approaches. The additional field level completes the extraction target classification. Field-level systems aim to extract single-slot records.

The last dimension to be used for classifying extraction systems is the page representation type. Many algorithms just tokenize given sources (on character level, word level,
phrase level, HTML tag and text level, etc.) for being able to execute the extraction task. Algorithms working with tokens generally go through the web page sequentially. In contrast to this, some algorithms use tree representations (e.g., DOM) for the examined web page. A tree representation offers much more information about the source to be analyzed but is also far more error-prone in cases when an HTML page is not formatted correctly. Finally, a web page can be examined visually by first letting a browser render it and then operate on information such as text box coordinates, etc.

The described criteria will be used to categorize all presented approaches with the supervision level as main criterion. An overview table is to be found in section 2.2.3. Since each presented system extracts data in its own manner, the previously introduced example will be continued. It is assumed that the data access layer successfully retrieved two pages about a digital camera and a camcorder as shown in Figure 2.11 and 2.12. The corresponding HTML representation is displayed below the rendered views.

**Manual Approaches**

Manual information extraction approaches for semi-structured sources are the most basic alternative for getting information out of web pages. In general, manual approaches work on the record or field level since extraction rules are created by a skilled user that adapts the applied wrapper to special targets. An example for a manual extraction approach is **TSIMMIS** [84] (The Stanford-IBM Manager of Multiple Information Sources), a framework for building web wrappers. The configuration of the framework’s wrapper is done via a specification file that states where to find the data of interest and how to transform it into the output format. Each specification consists of a triple of the form [variables, source, pattern]. The source describes which data is to be processed, the pattern specifies which parts are of interest in that source and the variables are filled with the extraction results. The data is provided by the Object Exchange model that allows the generic representation of structured information. For the web page in Figure 2.12, an appropriate specification file would contain “[“root”, “get(‘http://www.fedseeko.com/products/show/camcorder909)”, “#”]” for extracting the whole page’s content into the variable root, “[“ProductName”, “Product”, <h2>Product:<h2>]” for the product’s name, etc.

**XWRAP** [121] (XML-enabled Wrapper Construction) is another extraction system located on the edge of manually constructed and supervised wrapper approaches. By the use of formatting information and user feedback the system can identify regions of interest in a web page. In the running example, a region of interest would be identified by HTML.BODY.UL (although XWRAP mainly focuses on the HTML table element). Afterwards, an XML template is generated for extracting information from the page of interest. The template may contain rules, rule expressions, etc. for describing the extraction process. XWrap could be seen as a supervised approach since the user does not have to create configurations or write program code for extracting information. However, the user needs to have a basic knowledge about HTML and parsing for being able to provide the necessary feedback. Thus, the system is classified as a manual approach.
Product: Digi SLR 38

Specifications:
- Total Pixels: 10.75 MP
- Effective Pixels: 10.2 MP
- Height: 15 cm
- Length: 20 cm

Product: Digi Camcorder 909

Specifications:
- Optical Zoom: 48x
- PAL: yes
- Price: 299 €

Figure 2.11.: HTML Code of an SLR Camera Page.

Figure 2.12.: HTML Code of a Camcorder Page.
W4F [146] (World Wide Web Wrapper Factory) is a toolkit for building web wrappers programmed in Java. W4F implements the complete sequence of accessing, extracting, and integrating information. After having accessed a web page, the extractor creates a DOM representation of the page and extracts information using HEL (HTML Extraction Language) expressions. Finally, a set of mapping rules is executed to make the extracted data fit to the target application’s data model.

Other systems adopting manual approaches include Minerva [40] and WebOQL [9] (Web Object Query Language). WebOQL is a kind of query language for web pages. In a way, also XPath could be seen as a tool for executing manual information extraction. XPath provides all the concepts needed for accessing HTML tags and included text. In combination with XSLT [36] (Extensible Stylesheet Language Transformation), contents of web pages can be transformed into structured representations such as XML, only containing a set of desired information snippets. XPath and XSLT are heavily used in combination with each other and represent a standardized and clean way for transforming web pages into alternative representations.

Manual approaches were the first attempts at walking in the area of information extraction from semi-structured sources. As Table 2.3 will show, they are not subject to the current research in IE anymore. They are still of interest in cases where the extraction process must not be error-prone and retrieved results need to be correct in all cases. Naturally, this advantage goes along with a high labor input. Approaches with a higher degree of automation are of far more importance for this work since it cannot be expected from users of a federated product information system to write their own extraction routines. Thus, supervised systems are examined in the following.

Supervised Approaches

Supervised extraction approaches are able to generate wrappers automatically if a set of labeled training documents was provided by different users before. Generally, one can say that the bigger the set of training documents is and the more different users work on that set, the better are the resulting wrappers.

Rapier [27] is a supervised extraction system that uses a relational learning algorithm to extract information from job postings on the field level. The system is initialized with very specific extraction rules that are created for fitting the labeled data. An extraction rule consists of a pre-filler, a filler, and a post-filler pattern. The initial pre-filler pattern contains the tokens appearing in front of the field to be extracted (for the camcorder’s name in Figure 2.12, “<html><body><h2>Product: “) while the post-filler pattern initially holds all tokens behind the field (“</h2>...</html>”). The initial filler contains the information to be extracted (“Digi Camcorder 909”). The rules consisting of these fillers are successively replaced by more general rules to fit all labeled data in the input corpus. For pre-filler and post-filler it is taken into account that the tokens located close to the filler are generally the most important ones. Concerning the filler pattern, generalization can be done, e.g., by replacing the filler pattern with POS-tagging information or word counts describing the general structure of the target information. In the running example, a final rule could consist of the pre-filler pattern
Another supervised approach is **SRV** [69]. The user needs to annotate each target as positive example and each irrelevant information as negative example. SRV uses simple and relational features to describe the positive examples and to exclude the negative ones. A simple feature could indicate the length or the part of speech of the targets while a complex feature might describe neighbor elements. Thus, SRV’s approach is quite similar to the one of Rapier. It also works on the field level.

**SoftMealy** [95] uses the model of a Mealy machine to enable the extraction automatism. A Mealy machine is a finite-state transducer and belongs to the class of finite-state machines. In comparison to Moore machines, a Mealy finite-state machine’s output is not only dependent on its current state, but from a given input as well. This enables more compact representations of the machine than with the Moore notation. Being provided with a set of labeled documents, SoftMealy creates a body transducer for extracting tuples out of the web page and a tuple transducer for getting the attributes out of each tuple. Contextual rules created from the labeled documents identify the borders between different attributes and define the transitions between states of the Mealy machine. The Mealy machines can model each attribute permutation available in the pages and later on identify those attributes in new pages. SoftMealy works on the record level.

A different approach is taken by **STALKER** [129] (Supervised Learning Algorithm for Inducing Extraction Rules). Instead of targeting special web page fields independently from each other, STALKER expects the data of interest to be organized in a hierarchic way. Thus, basic rules for extracting simple or complex data objects out of complex ones are generated by the use of the labeled input documents. For the web page in Figure 2.12, the user would mark “Optical zoom”, “48x” as well as the other specifications to be extracted. STALKER then generates extraction rules. Each rule describes how to skip to the beginning and the end of the object in its parent object. If STALKER is working on the “<li>... </li>”-object, it would generate an extraction rule “SkipTo(<b>)” as well as “SkipTo(</b>)” for the specification keys (similarly with <i> and </i> for the values). If the examined object is a list (e.g., the “<ul>... </ul>”-object), the way to split the list into single objects is described in a rule as well. The utmost complex object is the web page itself where the first rules are to be applied. The embedded tree description created during the learning process shows how the different data objects are hierarchically related to each other. If the SkipTo method is not able to fulfill the rule description task, SkipUntil or NextLandmark (a landmark is a sequence of tokens) can be used. Additionally, when the landmarks cannot be described by deterministic strings, wildcards like “Number” or “HTMLTag” are available. STALKER acts on the record level.

Like STALKER, the authors of the paper about **DEByE** [113] (Data Extraction by Example) assume a hierarchic presentation of data in dynamic web pages and work on the record level. Attribute value patterns (AVPs) are used to identify the atomic data structures to be extracted from each web page. Like in most of the extraction approaches, the context of target data is used to create these patterns. In DEByE, the initial context is one token and is incrementally extended by the next token until the extracted AVPs
match the number of labeled AVPs. Later on, these AVPs are combined to create object extraction patterns (OEPs) which in turn may be composed to build higher-level OEPs. In the end, an OEP is represented in a tree-like structure and may be used for extracting data from new documents.

**WL²** [37] (Wrapper Learning System) is a supervised field-level approach that focuses especially on the extraction of tables and lists in HTML pages. The most important extension of WL² in comparison to other works is the usage of several document representations for information extraction. Valuable representations include DOM-level representations, token-level representations, and the rendered view of a page like it is shown in a browser. Classically, only one representation is to be used for the extraction approach. By combining all three representations and especially taking advantage of the rendered representation, WL² is more powerful than other approaches.

The central component of WL² is a master learning algorithm that is based on FOIL [144]. It consists of an outer loop for learning predicates and an inner loop for learning conjunctions. Both loops take the labeled document set as well as some builders as input. A predicate is a binary relation on spans (items to be extracted from a DOM tree are called spans by the authors). For example, a predicate \( p(s_1, s_2) \) may be true, if \( s_1 \) contains \( s_2 \) and \( s_2 \) is a li tag in a web page. Executing such a predicate on a web page yields all li tags contained in that page. A predicate can be created using different builders. A builder is based on one of the three document representations and offers two operations, namely, LGG (least general generalization) to find the most specific concept that covers all positive training examples and REFINE to create a combination of concepts to cover positive training examples in the case LGG is not possible. For example, a builder could be working on the token level and try to find the least general tokens located left and right of the information to extract. If the tokens left and right of this information differ too much, a combination of concepts has to be used. Finally, a conjunction of predicates is created that may be applied for extracting information out of given pages.

A quite recent approach for supervised wrapper induction on the record level has been introduced in the Pictor [190, 191] system. The authors justify the development of a supervised approach with the higher extraction quality. To keep the labeling process as comfortable as possible, two different techniques are applied. First of all, while the user labels attributes of records presented in the current web page, an incremental update process keeps the wrapper to be created by these labels up-to-date. This allows the system to guess attribute labels for other records in the page before the user actually examines them. Furthermore, previously created wrappers are used for guessing labels on pages to be labeled by a user. The authors evaluated that only about one tenth of the original labeling effort is needed with such support, thus allowing the creation of high-quality wrappers in less time.

The actual wrapper creation happens through the use of so-called broom structures. A broom’s stick represents the HTML path directing to the record to be extracted (in both, Figure 2.11 and 2.12, “html/body”) while the broom’s head includes the data region of interest, that is, the information to be extracted (in the example code, “h2”, “ul/li[0]/b”, “ul/li[0]/i”, “ul/li[1]/b”, etc.). Labeled web pages are used to create such brooms which
are clustered and finally generalized based on these clusters to create the actual wrappers (In the example broom heads, the indices would be removed). The resulting wrappers are then clustered by the host of their source web pages and organized hierarchically in DOM trees themselves using their tag paths. Thus, when a new page is to be analyzed, the wrapper framework can be searched easily for fitting wrappers.

Other approaches in the supervised IE area are WIEN [112] (Wrapper Induction for Information Extraction), WHISK [162], and NoDoSE [1] (Northwestern Document Structure Extractor). Supervised information extraction is far more elaborate than extracting information with manual IE approaches. However, the labeling of documents in advance is time-consuming since good extraction results can only be achieved with a high number of labeled documents. Furthermore, the presented approaches tie their extraction rules quite strongly down to the design of target web pages. Thus, the extraction of information from pages with completely different layouts would require the recreation of a sufficiently big document corpus and the relabeling of that corpus. Supervised approaches may therefore only be considered as a fallback alternative in the product information extraction routines to be developed in chapter 4. Semi-supervised or unsupervised approaches might fit better to the requirements of product information extraction.

**Semi-supervised Approaches**

The idea of semi-supervised information extraction approaches is to fully automate the extraction routine itself and not require any user input except some target web pages to be handled. Only after having created one or more wrappers and having extracted potentially interesting results, the user is asked for feedback to verify the correct execution of the wrapper induction. This feedback is used for further induction processes.

IEPAD [32] (Information Extraction Based on Pattern Discovery) is an example for a semi-supervised system. It consists of three components, namely, the extraction rule generator that accepts input Web pages, a graphical user interface called pattern viewer that shows discovered repetitive patterns, and the extractor module that extracts desired information from similar Web pages according to extraction rules chosen by a user. IEPAD works on the record level and thus heavily relies on repetitive patterns. A repetitive pattern is a substring at least occurring twice in a page. Since a web page may contain many useless repetitive patterns, IEPAD introduces the concept of maximal repeats. A repetitive pattern is a maximal repeat if not all pre-strings and not all post-strings of each pattern occurrence are identical. To find the maximal repeats, the web page content is translated into tokens and finally into a binary representation. By using so-called PAT trees, i.e., binary suffix trees, repeating sequences of tags or text elements are identified and used to generate a wrapper for the corresponding page. As this method only allows the identification of exact matches, the center star algorithm is used to enhance the extraction technique. Finally, the user is confronted with the extracted data. After having chosen valuable information out of the results, the generated wrapper can be used to extract information from similar pages automatically.

An example for IEPAD could be the following snippet from Figure 2.12: `<li><b>Optical`
Zoom:<i>48x</i> <b>PAL:</b>yes$. The dollar sign represents the end of the page. Each token is then translated into its binary representation, e.g., </li> to 000, </li> to 001, <b> to 010, </b> to 011, <i> to 100, </i> to 101, and any text token to 110. The complete HTML code in binary representation then looks like the following: 000|010|110|011|100|110|101|001|000|010|110|011|100|110|101|001$. Having this representation at hand, the PAT tree can be generated as presented in Figure 2.13.

![Figure 2.13: PAT Tree Representation for Extracts of HTML Code in Figure 2.12.](image)

Each suffix of the complete binary is included in the tree. Thus, it is possible to find repeating patterns and identify interesting regions for information extraction.

Two other semi-supervised extraction approaches have been developed in OLERA [31] (Online Extraction Rule Analysis) and Thresher [94]. Although working on the record level, OLERA allows the extraction of single records from a page and thus outperforms IEPA by this characteristic. However, since it demands a minimal user input describing roughly the region of interest on a given page, OLERA is only located on the border of supervised and semi-supervised systems. Thresher is similar to OLERA. It also demands some user input before generating wrappers. A quite nice feature of Thresher is that wrapper nodes are bound to semantic concepts using RDF.

Semi-supervised extraction systems are quite promising approaches since they automate the complete extraction task. After having created a wrapper, users only need to accept or reject the extracted data. Hence, a high quality of extracted information is assured while the level of interaction is still quite low. A combination of semi-supervised mechanisms and unsupervised mechanisms which are to be presented in the next section could yield desired results.
Unsupervised Approaches

Unsupervised information extraction systems are based on a fully automatic approach. Like semi-supervised systems, they try to identify repeating patterns either on the record or the page level. If a target information snippet is not presented in a repeating manner, such algorithms are generally not able to identify regions containing valuable information.

RoadRunner [41] is a page-level extraction system and based on an algorithm called ACME (align, collapse, mismatch, and extract). RoadRunner transforms the given web pages into a set of tokens (a token is either an HTML tag or text). Then, it defines the largest page to be the template for the extraction routine and compares this template with another similar web page serially. Each time the system detects a mismatch between the given pages, the template is generalized. There are two kinds of mismatches: string mismatches and tag mismatches. A string mismatch is treated as a dynamic data field containing information taken from a database. A tag mismatch is treated as an iterator or an optional depending on the content. This assumption is reasonable since database information included in a dynamic web page may be of different quantities (e.g., product search result lists in online malls) or not available for some entities (e.g., special prices for some products). Figure 2.14 shows an example of how RoadRunner executes the template matching algorithm. In the end, union-free regular expressions (UFREs) are generated based on the identified similarities and differences. The resulting template may then be used to extract information from other pages using the same template as well.

![Figure 2.14.: RoadRunner Example for the HTML Code in Figure 2.11 and 2.12.](image)

Lermann et al. [115] picture another approach for unsupervised information extraction from semi-structured sources. This algorithm groups extracted information into lists.
and tables with only very general assumptions about the structure of a web page. These assumptions state that the columns of table-like information contain similar data while rows can be modeled as complex data types being repeated in each row. Additionally, every such table-like structure begins and ends with a special string of characters. The algorithm first extracts the list or table by identifying the page template and calculating a number of features (separators and contents) for each extracted snippet. Then, columns are identified by classifying the extracted data. Finally, rows are identified by grammar induction on a sequence of class labels.

For identifying the page template, the algorithm compares pages that are based on the same template. Thus, this algorithm works on the page level. The shortest page is chosen as initial template (in contrast to RoadRunner which starts with the largest one). The tokens of this initial template are then concatenated until the template is complete. A template is called complete if all the elements of the original template have been removed that are not to be found in the other given web pages. The table-like structure can be identified by calculating the difference of a given page and the template.

For identifying table contents, special HTML tags and characters help to identify row ends. Each row is divided by defined separators that may as well be simple strings excluding the set “(,-).%”. In the end, the extracted information consists of all token sequences between identified separator strings. Additionally, data types contained in each row are compared with other rows to find out the row pattern with help of the DataPro algorithm. The assignment of text snippets to categories can be executed by AutoClass [33]. AutoClass is a model-based classification algorithm being able to identify the optimal amount of classes and the best mapping of extracts and classes. Finally, every data type ends up in a different class or column, thus retaining a structured copy of the dynamic data included in the web pages.

Another unsupervised extraction system is DeLa [180] (Data Extraction and Label Assignment for Web Databases). It operates on the record level and first locates data-rich sections in the HTML page by comparing DOM trees, before pattern suffix trees enable the actual information extraction.

A frequently cited approach concerning unsupervised information extraction is ExAlg [7] (Extraction Algorithm). ExAlg works on a hierarchic data model and supports optional elements as well as disjunctions. It uses the concept of large and frequent equivalence classes (LFEQs) to deduce a wrapper for similar web pages. Hence, it is also working on the page level. In a first stage, each web page is translated into a set of tokens (HTML tags and text elements). Each collection of tokens having the same frequency in all pages is then put into an equivalence class. The second stage is entered when the LFEQs cannot be grown anymore. Small equivalence classes (i.e., classes containing too few elements) are discarded since they might consist of the actual data to be extracted and thus do not belong to the page template. For the example pages in Figure 2.11 and 2.12, two LFEQs can be found ($\varepsilon_1 = \{< html >, < body >, < h2 >, ..., < /html >\}$ and $\varepsilon_2 = \{< ol >, < li >, < b >, ..., < /ol >\}$). The output template is created by integrating those LFEQs. The LFEQ holding elements only appearing once in every page is considered to be the root LFEQ (it contains inter alia the body tags of the page). After having determined empty and non-empty positions, the template is finished and may be used to
extract data from similar pages.

The next system to be presented here is called **DEPTA** [189] (Data Extraction Based on Partial Tree Alignment) and works on the record level. The authors of DEPTA divide the extraction process in two phases. The first phase consists of mining data regions and contained data records (MDR). This step is executed with the help of the MDR algorithm which has originally been presented by Liu et al. [119]. The new MDR-2 algorithm enhances the original mining process by using web screen scraping, that is, the HTML page is rendered by a browser engine and then analyzed by MDR-2 in a visual manner. Thus, the algorithm is independent from the way a web designer uses HTML tags to represent the repeating structures being filled with dynamic data. MDR-2 identifies data regions by drawing rectangles around them and then discovers data records inside all found data regions. The concept of generalized nodes is introduced to enable the combination of neighboring nodes to one data record. Finally, tag trees for each data region can be created.

The second step concentrates on the actual data extraction. For being able to get the relevant data out of each data record, the created tag trees need to be compared with each other. This step makes heavy use of simple tree matching (STM). Simple tree matching was first introduced by Yang [186] in 1991. STM is a simplified version of tree matching. A tree matching algorithm creates serialized representations of trees and then uses a mechanism similar to the Levenshtein algorithm [116] to compare those trees. Thus, pure tree matching allows level crossing of nodes and node replacement. Since the complexity of such tree matching algorithms is quadratic and level crossing as well as node replacement is not desired for the extraction approach, simple tree matching represents the better alternative in this case as it only allows a match between two nodes if their parents match. The maximum matching is the maximum number of pairs being equal for two trees and describes the similarity of two tag trees quite well. The STM algorithm is used for partial tree matching. That is, the tag tree with the maximum number of data fields is chosen as seed tree and compared with the other trees by STM. Every time an element is found in another tree that is not contained in the seed tree, it is extended. If the position of a potentially new element cannot be reliably determined, this element is skipped, thus justifying the term “partial tree alignment”.

The approach developed for **NET** [120] (Nested Data Extraction Using Tree Matching and Visual Cues) is based on the techniques introduced for DEPTA. Like DEPTA, NET works on the record level and uses visual information to enhance the extraction results quality. The main difference between DEPTA and NET is the additional opportunity to extract nested data records as well. NET traverses through the HTML tree in post-order to identify nested data records. Simple Tree Matching as proposed by Liu et al. is used here as well. Thus, subtrees of all nodes of the HTML tree are matched with each other. Hereafter, matched data items are aligned. In a final step all information is put into tables to be reused in other applications.

**ViPER** [160] (Visual Perception-based Extraction of Records) is a further approach working on the record level. The approach includes DOM-level, token-level, and visual information to discover records in a given web page. This is done by first comparing all subtrees of the web page’s inner nodes with each other. The comparison is achieved
by using an adapted version of Levenshtein’s edit distance. The original Levenshtein algorithm does not work well when comparing HTML trees since small changes might already result in big edit distances. Using thresholds is no good solution, too, as the threshold would be very low and many errors could disturb the overall algorithm. Therefore, the sequentialized subtrees are first examined for detecting tandem repeats. A tandem repeat is a non-empty sequence of elements that is encountered at least twice successively in a subtree. Such repeats are marked and ignored in the calculation of the edit distance since they are suspected to be data records. The comparison of two subtrees then returns an edit distance independent of the repeated appearance of data records.

Later on, the subtree comparison matrix shows paths for discovering data records. The whole process is supported by an additional visual analysis of the web page. That is, the page is rendered by a browser engine and boxes are drawn around the text elements. By comparing x- and y-profiles, the algorithm detects peaks and valleys that enable the partitioning of data records as well. Further heuristics like the distance of such regions from the center of the page are used to find out if the repeated structures are of higher interest for a potential user. In the end, also data alignment techniques are adopted to put the extracted data in corresponding rows and columns of a database table.

Álvarez et al. [5] offer a quite interesting approach for extracting data records from semi-structured sources in an unsupervised manner. The basic process consists of identifying a data region of interest, partitioning this data region into single records, and using multiple string alignment for extracting attributes from each record. Thus, the approach operates on the record level and is similar to the one of DEPTA. The algorithm is based on two observations stating that a record is a set of consecutive sibling subtrees and that contained record attributes of the same type have the same path from the web page’s root element. The first property additionally entails that the data region of interest can be identified by the common root node of all sibling subtrees contained in this region.

For finding the root node of the region of interest, all text nodes of a given web page are clustered based on their paths, that is, all text nodes with the same path are put into one cluster. In Figure 2.11, e.g., the cluster for “HTML/BODY/UL/LI/B” would contain the four product specification keys. Then, for every pair of text nodes in such a cluster, the deepest common ancestor is detected and its score is increased by one. The initial score for each node in the web page is zero. Hence, concerning the example cluster, both UL tags would receive a score of one while the BODY tag is scored with four. The node with the highest score is finally identified to be the root node of the region of interest (in this case, the BODY node). The region of interest is always the biggest list in the page. To assure that it is the correct list as well, a potentially entered query in a previous search step that caused the list to be generated can be searched in the text nodes to only take into account text nodes containing parts of the query when executing the clustering.

In the next step, different record candidate lists are created from which the correct one has to be chosen. A candidate division of the region of interest is created by clustering the subtrees of the found root node. Each subtree is translated into a string representation of HTML tag names and the generic TEXT element (e.g., “UL/LI/B” or “H2”). Then, by using a readjusted Levenshtein distance, all trees are compared
with each other (edit-distance similarity). The column similarity is used to cluster the subtrees into groups. The column similarity of two trees is high if they both share similar edit-distances with all other trees (the example contains four clusters: a = “H2”, b = “H3”, c = “UL/LI/B”, and d = “UL/LI/I”). Finally, the web page is translated into an abstract representation only consisting of a number of identifiers pointing to the subtree clusters (in the figure: “abcdcdcdcd”). Different candidate record divisions are created based on this representation (e.g., “a|b|cd|cd|cd|cd”). The candidate division offering the highest inter-record similarity is chosen as the final record list. Unique elements in the beginning or the end of the list are discarded (“a” and “b”).

As a last step, attribute values are detected. Therefore, each record is translated into a string representation and the center star algorithm (similar to IEPAD and DEPTA) is used to extract the attributes. That is, the longest record string is chosen as the master string and is extended with elements from other strings. Constant strings are assigned to the page template while varying strings represent the actual data.

Unsupervised information extraction approaches require no user input at all. Though this sounds auspicious, the quality of extracted information may suffer from the missing user interaction. Hence, as already mentioned before, a combination of semi-supervised and unsupervised approaches might produce optimal results. The next section will evaluate the different system concerning their suitability for the extraction task to be performed.

Comparison of the Approaches

All presented information extraction approaches for the domain of semi-structured sources can be categorized by the criteria described in the introductory section. These criteria include the supervision level (manual, supervised, semi-supervised, and unsupervised), the data level the approach is working on (field, record, page, and site), and the type of page representation used (tokens, tree, and visual). Table 2.3 displays all works with filled out values. They are ordered by the year of publication and extended with some short keywords about the central concepts employed.

The table permits some interesting conclusions. It can be seen that early approaches were manual or supervised systems. The first major approaches employing semi-supervised or unsupervised extraction were only published in 2001. This is no unexpected fact since semi-supervised and unsupervised systems are far more complex and initially needed some wrapper construction libraries to be able to push information extraction to the next level. Some novel publications nevertheless focus on supervised approaches (e.g., Pictor), motivated by extremely high extraction quality levels, thus allowing their serious application in web platforms. However, the superior number of publications for semi-structured information extraction in recent years proves that it is feasible to aim at the design of an unsupervised approach here as well. Furthermore, early works mostly used a token representation of the web page. Current works sometimes combine different page representations to yield better results. Such a combination of representations is promising and will be relevant for algorithms developed in chapter 4.

The information extraction task to be performed is mainly concerned with product
Table 2.3.: Comparison of Approaches for IE from Semi-structured Sources.

<table>
<thead>
<tr>
<th>Year</th>
<th>Author</th>
<th>System</th>
<th>Approach</th>
<th>Supervision</th>
<th>Data</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>Hammer et al.</td>
<td>TSIMMIS</td>
<td>Declarative Specs</td>
<td>Manual</td>
<td>Record Tokens</td>
<td>Page Tokens</td>
</tr>
<tr>
<td>1998</td>
<td>Califf et al.</td>
<td>RAPIER</td>
<td>Pre-Filler, Filler, and Post-Filler</td>
<td>Supervised</td>
<td>Field Tokens</td>
<td>Tokens</td>
</tr>
<tr>
<td>1998</td>
<td>Freitag</td>
<td>SRV</td>
<td>First-Order Induction</td>
<td>Supervised</td>
<td>Field Tokens</td>
<td>Tokens</td>
</tr>
<tr>
<td>1998</td>
<td>Hsu et al.</td>
<td>SoftMealy</td>
<td>Finite-State Transducers</td>
<td>Supervised</td>
<td>Record Tokens</td>
<td>Tokens</td>
</tr>
<tr>
<td>1999</td>
<td>Muslea et al.</td>
<td>STALKER</td>
<td>Hierarchic Rules</td>
<td>Supervised</td>
<td>Record Tokens</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>Liu et al.</td>
<td>XWRAP</td>
<td>XML Translation</td>
<td>Manual</td>
<td>Page Tree</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>Chang et al.</td>
<td>IEPAD</td>
<td>Binary Suffix Trees</td>
<td>Semi-sup.</td>
<td>Record Tokens</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>Crescenzi</td>
<td>Road-Runner</td>
<td>Template Generalization</td>
<td>Unsupervised</td>
<td>Page Tokens</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>Sahuguet et al.</td>
<td>W4F</td>
<td>HTML Extraction Language</td>
<td>Manual</td>
<td>Record Tree</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>Lermann</td>
<td>-</td>
<td>Template Aggregation</td>
<td>Unsupervised</td>
<td>Page Tokens</td>
<td>Tokens</td>
</tr>
<tr>
<td>2002</td>
<td>Laender et al.</td>
<td>DEByE</td>
<td>Context Analysis</td>
<td>Supervised</td>
<td>Record Tokens</td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td>Cohen et al.</td>
<td>WL²</td>
<td>First-Order Induction</td>
<td>Supervised</td>
<td>Field All</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Wang et al.</td>
<td>DeLa</td>
<td>Tree Comparison and Suffix Trees</td>
<td>Unsupervised</td>
<td>Record Tree</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Arasu et al.</td>
<td>ExAlg</td>
<td>Large and Frequent Equivalence Classes</td>
<td>Unsupervised</td>
<td>Page Tokens</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Chang et al.</td>
<td>OLERA</td>
<td>Drilling-down and Rolling-up</td>
<td>Semi-sup.</td>
<td>Record Tokens</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Zhai et al.</td>
<td>DEPTA</td>
<td>Screen Scraping</td>
<td>Unsupervised</td>
<td>Record Tree, Visual</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Liu et al.</td>
<td>NET</td>
<td>Tree Matching from Nested Rectangles</td>
<td>Unsupervised</td>
<td>Record Tree, Visual</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Hogue et al.</td>
<td>Thresher</td>
<td>Tree Edit Distance</td>
<td>Semi-sup.</td>
<td>Page Tree</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Simon et al.</td>
<td>ViPER</td>
<td>Tree Edit Distance and Coordinates</td>
<td>Unsupervised</td>
<td>Record All</td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Álvarez et al.</td>
<td>-</td>
<td>Subtree Clustering</td>
<td>Unsupervised</td>
<td>Record Tree</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Zheng et al.</td>
<td>Pictor</td>
<td>Incremental Wrappers</td>
<td>Supervised</td>
<td>Record Tree</td>
<td></td>
</tr>
</tbody>
</table>
specifications from producer websites. Such specifications are mostly presented in lists or tables. Thus, record-level extraction systems are generally applicable. However, producers often reuse their web templates for several products from one product family. Thus, also page-level systems might be of interest. The simple tree matching algorithm published by Yang helps in identifying similar pages for executing page-level algorithms.

The manual approaches presented above will not be taken into consideration since it cannot be expected from the user to create her own wrappers during information collection. Supervised approaches will only be interesting for vendor sources since the vast amount of producers is uncontrollable and the creation of an appropriate set of labeled documents is thus impossible.

From the remaining approaches, the most promising ones include ExAlg, DEPTA, NET, and ViPER. ExAlg offers a completely page content-independent algorithm that generates very good extraction templates. However, it only works well for at least three similar pages. It will be difficult to locate such an amount of product pages automatically which renders ExAlg unusable for this work. DEPTA, NET, and ViPER employ visual information which offers great help in identifying data records on producers’ product web pages. Additionally, the opportunity to mine nested data records with NET is valuable.

The extraction techniques of those works are rather universal on the one hand. On the other hand, they might not deliver the required quality for enabling product comparisons based on the extracted information. Thus, more sophisticated algorithms are needed for a better support of product specifications. These algorithms will be developed in the information extraction chapter.

Having accessed and extracted information from different sources, the next task of a federated information system is to integrate the extracted snippets with each other. Related work concerning this step will be presented in the following section.

### 2.3. Information Integration

The information extraction layer of a federated (product) information system provides relevant information in a structured way to the information integration component. However, the hierarchies and terminologies adopted in the different extracts may differ from each other heavily. Mechanisms for executing an adequate integration step are necessary. One possible concept for modeling the target hierarchy and terminology of extracted information are ontologies.

#### 2.3.1. Ontologies

Since the introduction of the Semantic Web, the term *ontology* has become a kind of buzzword for this area of the Web. Concerning the federated architecture to be presented, the ontology builds the basis of the product information integration part.

Originally, ontology is to be perceived as a philosophical discipline. The ontology deals with the being itself. It tries to represent concepts of reality using a natural or artificial language. Thus, a network of concepts with logical relations is created. Ontologists have always tried to model the whole world in one ontology. This plan has neither worked out
in the past, nor does it today. Ontological models can only cover a limited domain of the world’s knowledge.

In the computer science area, an ontology is the result of mapping such a knowledge domain (e.g., knowledge about products) to a predefined type of serialization, such as an XML dialect. The ontology is the most powerful of four different semantic models. The others are taxonomy, thesaurus, and topic map. Each model offers all semantic constructs of the previous one and extends this set by additional constructs [165]. A taxonomy only allows the representation of hierarchies. A thesaurus augments the taxonomy by offering the possibility to connect concepts using a similarity or synonymy relation. Topic Maps have been defined as an ISO standard based on XML [133]. They additionally offer the possibility of creating associations between concepts, adding properties to concepts, and referencing external documents.

Finally, the ontology is the most powerful of all four semantic models. Compared to topic maps, ontologies enable the description of relationships in more detail. A very powerful feature of ontologies is the division between schema and content. The schema describes, amongst other things, which concepts can exist in an ontology, how they may be related to each other, and which properties are available for a concept. The content of an ontology consists of instances that have to follow the defined schema. In general, an ontology schema is also referred to as terminology box (TBox) while the instances remain in the assertion box (ABox). The division into TBox and ABox is pictured in Figure 2.15. It is compared to the differentiation between table descriptions and table contents of a database as well as XML Schema and the XML content following such a schema.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Database</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
<td>TBox</td>
<td>XML Schema</td>
</tr>
<tr>
<td>Instances</td>
<td>ABox</td>
<td>Table Contents</td>
</tr>
</tbody>
</table>

![Figure 2.15.: Different Types of Schemas and Instances.](image)

Ontologies also allow to create inverse, symmetric, and transitive relations, handle multiple languages, and offer multiple inheritance. For being able to serialize ontologies, standardized ontology languages are needed. During the last years a series of such languages has been developed and standardized. The most important ones are listed in the following.

**Ontology Languages**

The major promoter of the Semantic Web is the World Wide Web Consortium. Hence, some standardized serialization languages for describing ontologies have been recommended by the W3C.
These include the Resource Description Framework [93] (RDF), a set of standards for describing the elements of an ontology ABox. It consists of two basic components, namely, resources and literals. A typical RDF statement (RDF triple) is built by a subject, a predicate, and an object. The subject and the predicate are always resources that can be identified by a URI. The object may either be a resource or a literal. Depending on the type of the object, the predicate describes a property or a relation. An RDF document is a collection of such RDF triples with statements about corresponding resources and can also be called an RDF graph. Additionally, the Resource Description Framework Schema [24] (RDFS) enables the description of an RDF ontology’s TBox. An introduction to RDF and RDFS can be found in Introduction to the Semantic Web and Semantic Web Services [188] by Liyang Yu.

RDFS is only able to describe simple ontology vocabularies and does not allow the usage of all available ontology concepts by far. Therefore, the Web Ontology Language [15] (OWL) has been introduced by the W3C. Its major extension consists of a set of operators and quantifiers. Furthermore, properties have been explicitly divided into object properties and data type properties. OWL is available in different profiles. Reduced profiles remove some of its expressiveness to guarantee computational completeness and decidability. An introduction can be found in the previously cited book by Liyang Yu.

RDF(S) and OWL are ontology languages defined by the W3C with a strong focus on the Semantic Web. In closed systems where efficient inferencing algorithms are of importance, other ontologies might be useful as well. One of those is Frame Logic [109] (F-Logic or FLo). It was originally developed for deductive databases. However, today it is mainly used for representing ontologies. The authors of the original paper describe F-Logic quite well as standing “in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming”. One of the main differences between F-Logic and RDF(S)/OWL is the so-called closed world assumption. That is, F-Logic expects every statement that is not given to be false. In RDF and OWL, a statement that is not given is unknown (open world assumption). Additionally, F-Logic is undecidable while the OWL DL-profile is decidable as depicted in the previous section.

When ontologies are managed in a semantic repository, a machine or a human being needs to be able to access these ontologies in a similar way like databases are accessed with SQL. The SPARQL Protocol and RDF Query Language [142] (SPARQL) is such a graph-based query language for accessing data sources managing their data as RDF. It offers possibilities to query graph patterns using conjunctions and disjunctions. A SPARQL query may return result sets or RDF graphs. Since its purpose is the same, SPARQL’s syntax is very similar to the one of SQL, thus easing the learning process heavily for people already being used to SQL. The SPARUL Protocol and RDF Update Language [154] (SPARUL) is an extension to SPARQL for adding, updating, and deleting RDF in a semantic repository.

Ontology Types

As described in the previous section, ontologies offer a reusable and extendable way to represent knowledge from a given domain. The central idea of ontologies on the Semantic
Web is to share such knowledge with other people and semantically connect different information systems by using the same semantic model.

Existing ontologies can be separated into upper ontologies, domain ontologies, and application ontologies. Upper ontologies contain very general concepts, such as *thing* or *time*. As the name suggests, they build the upper framework of the knowledge domain to be modeled and can be reused in virtually any use case. Domain ontologies concentrate on a concrete knowledge domain (e.g., health, movies, or shopping). As already mentioned before, ontologies cannot model all concepts in the world. Therefore, domain ontologies try to concentrate on one concrete area while still remaining reusable. Finally, the third ontology class, namely the one of application ontologies, comprises ontologies with a very low reusability. In general, application ontologies are adapted to a concrete use case in a dedicated information system to enable high performance while not caring about the possible reuse of modeled knowledge. The connection of all three ontology types enables the creation of a complete semantic model for a specific application. The described hierarchy of such a model is pictured in Figure 2.16.

<table>
<thead>
<tr>
<th>Upper Ontology</th>
<th>Abstract</th>
<th>Highly Reusable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Ontology</td>
<td>Concrete</td>
<td>Reusable</td>
</tr>
<tr>
<td>Application Ontology</td>
<td>Concrete</td>
<td>Not Reusable</td>
</tr>
</tbody>
</table>

Figure 2.16.: The Ontology Hierarchy.

Naturally, upper ontologies only make sense if they are publicly available for everybody and if their number is not too big. A widely known upper ontology is the Descriptive Ontology for Linguistic and Cognitive Engineering [81] (DOLCE). DOLCE contains classes such as *space-region*, *non-physical-endurant*, or *feature*, thus proving the described characteristics of an upper ontology. Other upper ontologies are IDEAS [80] and COSMO [28]. Upper ontologies are a controversial concept. Apart from the fact that an upper ontology represents only one of many different ways to divide the world into concepts (there is no self-evident way to divide the world into concepts), the use of such an ontology is questionable. Any application working with ontologies will strongly focus on the domain it is located in and possibly applied reasoning and inference mechanisms will mostly stay on the level of the corresponding application and domain ontologies. Hence, in many cases, an upper ontology is a beauty-related extension for a semantic information system, causing a non-neglectable overhead in managing ontology contents.

Domain ontologies exist in a much larger number. They are not always available to the public. As this work focuses on product information, the next section will be dedicated to product domain ontologies. Application ontologies exist in vast amounts as almost every semantic system uses its own. They will not be discussed here since the application-specific design hinders their deployment in the federated semantic product search environment to be examined during the next chapters.
Product Ontologies

As mentioned earlier, product ontologies are important for this work as they might be employed for managing collected product information in the federated architecture to be developed later on. One of these ontologies is eCl@ss [55]. eCl@ss is a hierarchical system for grouping materials, products, and services including specific properties of these concepts. The hierarchy has four levels, namely segments, main groups, groups, and commodity classes, thus identifying each element by a four-digit number. eCl@ss is available in different natural languages. The original definition of eCl@ss is not bound to any ontology language. However, Martin Hepp has created an OWL version of eCl@ss [88] called eClassOWL, thus enabling the usage of the product information model in a semantic information system. A minimal example for our Digi SLR 38 camera describing the available resolution is presented in Figure 2.17.

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:gr="http://purl.org/goodrelations/v1#"
  xmlns:fed="http://www.fedseeko.com/products/
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:eco="http://www.ebusiness-unibw.org/ontologies/eclass/5.1.4/#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <eco:C_AKN885002-gen rdf:about="fed:digiSLR38">
    <eco:P_BAF559001>
      <gr:QuantitativeValueInteger rdf:about="fed:QuantitativeValueInteger_1">
        <gr:hasUnitOfMeasurement rdf:datatype="xsd:string">
          Effective Pixels
        </gr:hasUnitOfMeasurement>
        <gr:hasValueInteger rdf:datatype="xsd:integer">
          10750000
        </gr:hasValueInteger>
      </gr:QuantitativeValueInteger>
    </eco:P_BAF559001>
    <eco:P_BAG073001 rdf:resource="eco:V_BAC386001"/>
  </eco:C_AKN885002-gen>
</rdf:RDF>
```

Figure 2.17.: Representation of Digi SLR 38 with GoodRelations and eClassOWL.

Another ontology developed by Hepp is ProdLight [90]. In contrast to eClassOWL, ProdLight is a very small ontology and does not try to model the whole product domain. Instead, a minimal set of statements is used while the identification of products is done by referencing eCl@ss concepts or the United Nations Standard Products and Services Code [122] (UNSPC). Finally, also GoodRelations [91, 92] was developed by Hepp. GoodRelations aims at describing the relationships between web resources, offerings presented on these web resources, legal entities, prices, terms, and conditions, and an
ontology describing products and services like eClassOWL. It is written in OWL and designed for practical use in online malls, such as the ones characterized as vendors in previous sections. GoodRelations is used in the given example as well for creating product attributes by the help of generic properties like hasUnitOfMeasurement.

After having built a broad overview of the ontology domain, the following section concentrates on one important application field of ontologies, namely, ontology matching.

2.3.2. Ontology Matching

The third step to be performed in a federated information system includes the integration of extracted information deriving from the same domain but being represented using different schemas. For example, producers of digital cameras might describe the same specifications of a camera using miscellaneous terms. It cannot yet be expected to always find an ontology for such information online that describes the information schema. However, in template-generated pages each web source’s information has a distinct structure defined by some internal schema. It is reasonable to assume that extracted information (e.g., product specifications) is implicitly modeled by an ontology. Since the federated product information system to be developed in chapter 3 bases its information management on ontologies as well, the integration task resides in the area of ontology matching.

Ontology matching consists of solving issues like integrating several concepts from a source ontology into one concept of a target ontology, adding or removing properties in one ontology to fit to the counterpart’s property set, changing a property’s value to use the same language or unit, etc. The arising problems are manifold. According to [34], the ontology matching process can be of three different types. If, for example, an application ontology is to be matched with a domain ontology, the type is local ⇔ global. If the application ontology of one system is to be integrated with another system’s application ontology, the type is local ⇔ local. Finally, if two systems’ ontologies are to be integrated permanently, a new domain ontology can be created and the type is local ⇒ global.

The standard approach to bring two ontologies together can be described by three concepts, namely ontology matching, ontology alignment, and ontology mapping. Ontology matching is also used as an umbrella term for the whole process. The following definitions are partly taken from [58].

**Ontology Matching.** Ontology matching is the process of finding relations or correspondences between entities of two or more ontologies.

**Ontology Alignment.** An ontology alignment is a set of rules describing how to integrate different ontologies with each other. An alignment is therefore the result of the previous ontology matching step.

**Ontology Mapping.** Ontology mapping is the directed execution of an ontology alignment to map elements of the source ontology on corresponding ones of the target ontology.

Advanced ontology elements (e.g., rules) are generally not taken into account in ontology matching, hence, algorithms for matching databases or XML are often quite similar to
the ones of ontology matching. Actually, all kinds of matching techniques for different knowledge representations including ontology matching could be subsumed by the term schema matching. Schema matching techniques can be divided by the employed matching information base being the schema or available instances (Figure 2.15). Since the term schema matching is used as a general name for integrating knowledge representations (independent from the employed matching information base), the attributes schema-based and instance-based will explicitly refer to the matching information base.

According to the explanations above, ontology matching can be divided into TBox-based matching and ABox-based matching. TBox-based matching tries to integrate two ontologies by only using information available in their TBoxes. ABox-based matching uses instance data for the matching process since an ontology’s TBox does often not provide enough information for integrating it with another one. Approaches from different schema matching areas (database integration, XML matching, etc.) can often be transferred to the ontology domain with only little effort. Some authors overlook this fact which led to a number of publications offering ontology matching concepts which are quite similar to existing ones from other schema matching areas.

To give an overview for the schema matching process, some general, knowledge representation-independent matching properties being relevant for most of the approaches will be pictured in the following section.

The Matching Process

The characteristic schema matching sequence consists of the actual matching step, an aggregation step, and a selection step. Figure 2.18 displays each of those steps roughly showing the inputs and outputs of corresponding components.

State of the art matching systems usually apply several elementary matchers for finding an alignment in the first step. A matcher calculates similarities of different nodes for two given input models $S_1$ and $S_2$. The output of such a matcher is a similarity matrix
$M_i$ containing similarity values for each pair of nodes from $S_1$ and $S_2$. Depending on whether the system executes its matchers sequentially or in parallel, the overall matcher is called hybrid or composite, respectively. Determined by the matcher type $i$ that was executed before, the resulting matrix values may have been calculated by different kinds of similarity functions.

In related work, numerous matchers have been developed. Those matchers can be categorized by a number of properties. A detailed matcher categorization is given in [157]. Figure 2.19 is taken from this survey as well and shows the available schema-based matching techniques.

![Schema-Based Matching Techniques](image)

Figure 2.19.: Schema-based Matching Techniques. [157]

Considering their matching granularity, matchers can generally be divided into element-level and structure-level matchers. Element-level matchers only work on the elements themselves, that is, they ignore the position of the elements in the schema tree. In contrast, structure-level matchers use the whole schema tree to categorize the current element and calculate a similarity to other elements, e.g., by checking the element’s children or parent nodes. Element-level matchers can further be divided into syntactic element-level matchers and external element-level matchers.

Syntactic element-level matchers only use information contained inside the given elements for calculating similarities. For example, a syntactic element-level matcher may analyze an element’s name in $S_1$ to compare it to another element from $S_2$ using the Levenshtein distance. Such a matcher would be string-based. String-based matchers are applied in nearly every matching system. Other syntactic element-level matchers can be language-based (using NLP techniques for identifying individual words or phrases or execute a morphological analysis on given element strings) or constrained-based (e.g., to
detect similar data types like day and workingday or multiplicities of an element).

The second class of element-level matchers are external element-level matchers. Such matchers use additional information sources for calculating similarity values. Linguistic matchers are based on lexicons or thesauri such as WordNet, e.g., for retrieving synonyms of given schema strings to extend the element’s information base. Alignment-reusing matchers check previously created alignments for detecting similarities (e.g., both input schemas have already been matched with a third schema, hence, allowing a transitive mapping). Matchers based on upper ontologies use knowledge models like DOLCE to gain additional information about an element and eventually detect similarities to other elements.

Structure-level matchers can be divided into syntactic, external, and semantic structure-level matchers. Their different subtypes may use graph matching (graph-based matchers), taxonomic structures (taxonomy-based matchers), metadata about a schema’s structure gained from a structure repository (matchers based on structure repositories), or description logic (model-based matchers) to calculate similarities. For different reasons to be explained in section 5.3.2, structure-level matchers are not that important here.

Figure 2.19 states that all of the presented matcher types are schema-based matchers. However, if instances of a given schema are available, they may contain valuable information and should be included in the matching process as well. In general, all matcher types can be applied on documents containing instances, too. But, since the instances’ structure is also encoded in the corresponding schema, element-level matchers are better qualified for performing instance matching.

After having created a set of similarity matrices (the similarity cube) through corresponding matchers, an aggregator may consolidate the different matrices \(M_i\) into one overall matrix \(M\) (second step in Figure 2.18). Like for the matching algorithms, there is a wide set of aggregation strategies available. Typical representatives are average and \(\max\). The average strategy calculates the average similarity of all matrices' similarities for a particular element pair while the max strategy transfers the maximum similarity value of an element pair to the overall similarity matrix \(M\).

Having an overall similarity matrix \(M\) at hand, the selection step can be executed (third step in Figure 2.18). The output of this step is a matrix \(M'\) describing the resulting schema or ontology alignment. \(M\) still includes similarity values for all possible element pairs of the input documents. The employed selector removes the biggest part of these similarity values such that only mappings with a high matching similarity value remain. Typical strategies are \(\max N\), threshold, or \(\max \Delta\). \(\max N\) only accepts the N best similarity values as mapping rules, threshold accepts all similarity values above a certain threshold while \(\max \Delta\) chooses the highest matching similarity as well as all similarity values close to the maximum value.

The classical sequence of matching, aggregating, and selecting is often diversified by newer systems [141] to offer more flexibility. However, the first step in every matching system is always comprised by a set of matchers while the last step is always a selection process. In between, the order may vary. For example, an aggregator may receive its input from several selectors or a set of matchers and other aggregators. The different approaches for adopting matchers, aggregators, and selectors will be presented in the
following. Since it is common practice to divide the approaches by their matching information base (schema and/or instances), this will be done here as well. A final section gives an overview on all mentioned approaches.

The section about information extraction described how to gain valuable information from differently structured sources. Thus, the employed example will be continued throughout this section based on the simple schemas A and B presented in Figure 2.20.

![Diagram](image-url)

Figure 2.20.: Example Schemas from the Digital Camera Domain.

**Schema-based Matching**

Schema-based matching is the most common type of schema matching. Systems from this category only employ the information model and do not account for available instance data.

**Cupid** [123] was the first hybrid schema-matching system being based on a structured categorization of elementary matchers mainly taken from **Semint** [117] (Semantic Integration of Heterogeneous Databases), **DIKE** [137], and **ARTEMIS** [17]. It uses a combined linguistic matcher as well as a structural matcher.

The linguistic matcher first normalizes the given schemas’ elements, that is, it tokenizes the element names (e.g., “Digital”, “Photo”, and “.” in Figure 2.15), expands existing abbreviations (“Photo” to “Photography”), and eliminates insignificant tokens hereafter (remove “.”). Thesauri are adopted throughout this step. Then, schema elements are categorized by clustering them with respect to data types, schema hierarchy, and linguistic content. Finally, the elements can be compared with each other. The structural matcher assigns leaf elements an initial similarity based on the linguistic similarity and a similar set of leaf neighbors and calculates the structural similarity of other elements bottom-up by comparing these nodes’ leaf sets. In the figure, first DigitalPhoto. and FilmPhoto. would be compared with DigitalCameras and AnalogCameras before Photography and Cameras can be matched with each other. Contrariwise, leaf elements can receive a higher structural similarity if their ancestors have high structural similarity values. For
being able to process arbitrary schema graphs, Cupid is able to transform a rooted graph into a tree by copying sub-trees. Thus, Cupid may operate on arbitrary models, such as relational schemas, XML Schema, or OWL.

**COMA** [48] (Combination of Matching Algorithms) is a framework for the creation of composite schema matchers. It includes a component for evaluating matching results as well. Six elementary matchers, five hybrid matchers, and one alignment reuse matcher are included in the framework. The alignment reuse matcher is of special interest since COMA was the first system to offer such a matcher type. It was enhanced by COMA++ [11].

COMA may operate on arbitrary schemas as long as they can be imported into the internal representation. Then, a user may give some optional input for the schema matching process. Several matchers are executed on the two input schemas hereafter. The elementary matchers use affixes (e.g., the prefix “Digital” in Figure 2.15), n-grams, soundexes, edit distances, synonyms, or data types of the schema element names. The hybrid matchers are based on simple matchers and can compare element names in a more elaborate way, e.g., by extending element names with the elements’ paths (for example, Photography/DigitalPhoto. in Figure 2.15). Hybrid matchers based on the schema structure are included as well, e.g., for comparing the sub-nodes of different elements. The reuse-based matcher has access to a matching repository which can be employed for transitivity detecting matches. Depending on the user support, the matcher combination may vary and can change iteratively. If no user input is given, a standard matcher configuration is adopted. The matchers’ resulting similarity cube is combined by using different aggregation strategies (max, weighted, average, or min) and finally alignments are selected through corresponding selection strategies (maxN, maxDelta, or threshold).

**Similarity Flooding** [126] is a hybrid schema matching system and uses constraint-based matchers while doing fixpoint computations on the graph representations of its input schemas.

The basic approach consists of the following steps. In the first step, given schemas are imported and transformed into an internal graph representation (similar to COMA) to be independent of the original schema representation (SQL, XML, OWL, etc.). Then, an initial matching map is created by using a simple string matcher that compares schema elements based on affixes. The initial similarities are fed to the actual similarity flooding component. This component refines the given mappings by increasing pair similarities based on the similarities of the pair elements’ neighbors. An example for the photo camera domain is shown in Figure 2.21.

A pairwise connectivity graph is created that contains temporary element pairs and connections to other element pairs if the original elements of the first pair had the same relation to the elements of the second pair in the original graphs. The final similarity propagation graphs allow spreading similarity values to other pairs. The less pairs are connected with a pair, the more similarity they provide to the other one. After iteratively calculating the final similarity values, a selection is executed in the fourth step. Different algorithms from the domain of bipartite graphs can be employed here (e.g., the stable marriages algorithm). The preferred algorithm changes the original similarities to relative
similarities and only accepts similarity values above a certain threshold.

In S-Match [77] (Similarity Match), the authors tried to implement a semantic matching approach. They first defined the concept of/at a label as well as the concept of/at a node. Since the label of a node in a schema tree has some semantic meaning in the real world, it can generally be assumed that the concept of/at this label consists of the set of documents concerned with the label’s meaning. The concept of/at a node is the set of documents that would be classified under this node.

The taken approach consists of four steps. In the first step, for all labels the concepts at the labels are computed. This is done by tokenizing the labels, lemmatizing each of the tokens (in Figure 2.20 “Cameras” could be lemmatized to “Camera”), using WordNet to get different senses of the lemmatized tokens, and rebuilding the complex concepts out of the label tokens and WordNet senses. In the second step, the concepts at all nodes are computed. For this task, the whole path from the root element to the current node is taken into account. In step three, label pairs for the given input schemas are built and their semantic relations are computed by using simple element-level matchers. The final step calculates the semantic relations among concepts at nodes through structure-level matchers that are based on the matching results of the element-level matchers executed in step three. The S-Match system is built in a modular way and offers around 20 element-level matchers. For matching on the structure level, SAT resolvers and reasoning methods are available.

The first schema matching approaches introduced a multitude of matcher types for integrating given schemas with each other. Current works focus on the flexible combination of matchers developed in previous works for automatically fitting certain domains or even single element pairs of the input schemas. One of these approaches is MatchPlanner [50] which is based on decision trees for gaining optimal matching results. The authors of MatchPlanner state that it is not efficient to simply execute a big set of matchers on
each available element pair and then combine matching results through an aggregation function. First of all, it impairs the performance since for two schemas with \( n \) and \( m \) elements, respectively, and a set of \( k \) matchers \( n \times m \times k \) similarities have to be computed. Furthermore, the matching quality might suffer, e.g., if many string matchers are executed on a pair of syntactically different but semantically equivalent elements. Eventually, the aggregation of similarity values often needs to be tuned by hand which makes such approaches quite unflexible.

A decision tree as employed in MatchPlanner allows the controlled execution of matchers in a hierarchic manner. An example of such a decision tree is given in Figure 2.22.

![Decision Tree](https://via.placeholder.com/150)

**Figure 2.22.: Example for a Decision Tree of MatchPlanner. (inspired by [50])**

In the example graph, the elements *DigitalPhoto.* and *DigitalCameras* from Figure 2.20 would be first checked for equality by a simple string comparison matcher. Since the elements are not equal, their label size sum needs to be computed. It is smaller than eight, thus the Levenshtein similarity would be calculated (= 1 − \( \text{levenshtein\_distance} \), normalized by the longer string). As this similarity is greater than 0.1, the elements are accepted to be a match and are part of the final alignment. The example shows that the matchers’ results along the decision tree’s matching path decide about which follow-up matchers should be applied. The similarity of the examined element pair is given by the last executed matcher. Decision trees can easily be adapted for the scenario they are located in. Therefore, a standard decision tree is taken from the decision tree library and is iteratively adjusted to fit the scenario based on the user’s feedback concerning generated alignments.

An algorithm presented by Akbari et al. [4] uses two steps for creating an ontology alignment. In the first step, ontology elements are compared using a lexical matcher. The matcher tokenizes input strings by non-alphabetical characters (in Figure 2.20 this is either the blank character or a dot). Then, all tokens being equal in both strings are removed (e.g., “Digital” would be removed for the pair *DigitalPhoto.*, *DigitalCameras*). After having concatenated the remaining tokens, the Levenshtein distance is used for comparing the strings. The lexical matcher is executed on classes, object properties,
data type properties separately, thus, three similarity matrices are generated. The second step is the execution of a structural matcher. This matcher first creates neighborhood matrices containing a one if two elements are neighbors and 0 otherwise. Based on these neighborhood matrices a grid is created for each node. Such a grid is again a matrix describing the neighborhood of one schema element in detail. The first column of the grid contains for each direct neighbor of the current node the number of neighbors it has. Thus, each row of the grid corresponds to one of the element’s neighbors. Consequentially, the other columns hold the amounts of nodes being neighbors to the node of the current grid row. For the element Cameras from Figure 2.20 the grid would be \([2, 2, 2], [2, 2, 2]\) since Cameras has two neighbors (two rows) and each of them has two neighbors again (two additional columns). For creating the structural similarity matrix, grid rows of the first schema are compared to grid rows of the second schema. Thus, the most similar neighbors can be detected structurally and build a new candidate for the final alignment. The structural similarity matrix is tuned by increasing similarity values of neighbor pairs in case the original element pair is similar or by increasing the similarity if two nodes have common data type properties. A weighted combination of both matchers creates the final similarity matrix.

Schema-based matching is the most common type of integrating knowledge representations with each other. Since valuable information may also be included in corresponding instances, in the following, systems directly working on instance data are described.

**Instance-based Matching**

Instance-based matching has been adopted in a set of different approaches. In contrast to schema-based approaches, it can often not be expected to recognize the complete information model through the given instance data. In many cases, it is only included implicitly. This is a major drawback. However, since instance data is often provided in a large scale, it enables the use of machine learning techniques for generating alignments. Some of the available instance-based matching systems are presented in the following.

**Automatch** [19] is solely based on instance data of different schemas. The instance data is compared with an internal knowledge base using techniques of the machine learning domain, especially Bayesian learning. This knowledge base is called the attribute dictionary. The dictionary contains schema elements and their possible values together with probability estimates for these values. The dictionary is created by analyzing instance data, extracting existing values for each of the encountered schema elements, and creating a significant feature set for each element. In our example (Figure 2.20), the instance DigiSLR38 and many other such instances would be examined to create a feature set for DigitalCameras as well as DigitalPhoto. (for this example, instances would only be connected to concepts of one schema). Having the dictionary at hand, two input documents containing instance data can be matched with dictionary elements based on their element values. The final alignment is calculated transitively by connecting matchings from one instances document to the dictionary and from the dictionary to the other instances document. Similarity values are summed up to create the overall similarity value for an element pair.
GLUE [49] is a system for semi-automatic taxonomy and ontology matching. It is mainly based on joint distribution probabilities of concept instances for predicting matchings. Several techniques like machine learning (especially Bayesian learning like in Automatch) and relaxation labeling are employed for identifying the final alignment.

The first component to be executed in GLUE is a meta-machine-learner. Its goal is to determine joint distribution probabilities of concepts from both taxonomies. These joint distribution probabilities help to calculate the similarity of two concepts. For example, the joint distribution probability $P(DigitalPhoto., DigitalCameras)$ is to be calculated by counting the number of elements being instances of $DigitalPhoto.$ and $DigitalCameras$. Other probabilities to be calculated would be $P(DigitalPhoto., DigitalCameras)$, $P(DigitalPhoto., DigitalCameras)$, and $P(DigitalPhoto., DigitalCameras)$. The meta-learner employs two elementary learners for these tasks, namely, the content learner and the name learner. The content learner divides model $A$'s instances into instances of $DigitalPhoto.$ and other instances. Then, based on intrinsic features of each instance (element name, attribute values, etc.), it trains itself for detecting instances of $DigitalPhoto.$. Instances of $DigitalPhoto.$ can later on be detected in the instances of model $B$. The same task is executed the other way around. The name learner works similarly, just basing its feature detection on the complete path of an instance from the schema’s root tree to the instance itself. The results of the content learner and the name learner are combined in the meta-learner.

The similarity estimator uses the meta-learner’s results to estimate similarities between each of both taxonomies’ elements (e.g., by using the Jaccard coefficient). Finally, the relaxation labeler detects the best alignment for both schemas. For example, different heuristics and rules are applied to detect the best mappings. A rule might state that if all instances of $DigitalPhoto.$ are recognized as instances of $DigitalCameras$, then also $DigitalPhoto.$ should match $DigitalCameras$. The outcome of GLUE is a set of 1-to-1 mappings.

If schema and instance data is available, a matching system should use both data structures conjointly for creating alignments. Such combined matchers are to be presented in the next section.

Combined Matching

Schemas as well as instances provide valuable information for a potential matching process. For example, the information model could be used to create initial alignments which are to be refined using knowledge gained from instance contents. The following systems use the best of both worlds.

OLA [59] (OWL Lite Aligner) follows a similar approach as developed for Similarity Flooding. It is specifically based on OWL and uses so-called i-th level contributors for adjusting similarities of node pairs.

For being able to work on the input data, OLA generates a graph-based version of the given OWL files. It does not take advantage of the standard RDF graph representation, but tries to model all relations available in the OWL data explicitly using a representation format called OL-graphs. For example, if an owl:allValuesFrom statement is given, OLA
creates a connection between the corresponding nodes attached to this statement. Then, for accessing initial similarity values, the 0-th level contributors are used. 0-th level contributors of a node pair (DigitalPhoto, DigitalCameras) are the attributes of each of these nodes, e.g., URI references or labels (in this case, the labels “DigitalPhoto” and “DigitalCameras”). Their similarities are calculated by simple affix matchers. Then, each node pair contributes similarities to its neighbor node pairs. The final alignment may be chosen using a threshold function that only allows the best matching values to create mappings. As mentioned before, the idea of OLA is quite similar to Similarity Flooding, mainly being adjusted to use particular ontology features.

In the course of the SWAP project (Semantic Web and Peer-to-Peer), NOM [53] (Naïve Ontology Mapping) has been developed. As the name already states, it is based on naïve rules (R1-R17) to calculate similarities between ontological entities. All of the applied rules are created before the actual matching process intuitively. Since they only represent very general statements, they can be employed for matching arbitrary ontologies. Some example rules are R1 (If two entities have the same or quite similar labels, they are equal; this is the case for “Digital Photo.” and “Digital Cameras” in Figure 2.20), R5 (If super-concepts of two concepts are equal, these concepts are similar; in our example, DigitalCameras could be detected as a matching partner for DigitalPhoto if Photography and Cameras have been matched before.), and R10 (If two concepts have the same instances, the concepts are equal; DigiSLR38 instantiates DigitalPhoto and DigitalCameras and thus is a secure hint that DigitalPhoto and DigitalCameras are equal.).

For creating the alignment, first independent similarities are calculated. R1 and R10 are independent since they are not based on other mappings. Then, dependent similarities like R5 can be computed. The resulting similarity values are aggregated, e.g., by using a sigmoid function that assigns a high weight to high similarity values and a low weight to low values. Finally, the selection step takes place which chooses the optimal mappings by setting a certain constant threshold or by selecting the best similarity value and accepting all values close to this value.

The weak point of NOM as well as most of the other matching approaches is the performance. In general, the focus lies on effectivity and not efficiency, thus leading to horrible runtimes for matching algorithms. This problem was tackled during the development of QOM [52] (Quick Ontology Mapping), the successor of NOM. QOM fully focuses on efficiency while only having small loss regarding the alignment quality. This is accomplished by using many heuristics for selecting matching candidates. Standard approaches generally try to match all elements of one ontology with all elements of the other one. QOM’s strategies for discarding some of the possible candidate mappings include random (randomly leave out some candidates), label (only try to match candidates with similar labels), area (having some mappings at hand, only choose candidate mappings around these mappings), change propagation (try to match neighbors of pairs that were mapped in the last iteration), and hierarchy (start from the top-level elements to find mappings). A typical combination strategy first applies the label strategy, then the change propagation strategy, and finally the random strategy if the previous strategies did not find enough mappings.
ASID [22] (Another Schema Integration Daemon) is a relational database schema mapper. The authors state that from the broad set of available matcher types, not all matchers are to be considered as powerful and trustable similarity measure sources. Indeed, some matchers, especially instance-based matchers that may be mislead by dirty database contents, can introduce uncertainty in the matching process. Thus, matchers are divided into strong and weak matchers.

The first step in ASID is to load the schema information into an internal representation like it is done in most matching approaches. Then, strong matchers are executed on the schemas. In ASID’s implementation only two strong matchers have been included. The first one is a name matcher that uses the Jaro metric for calculating similarities. The Jaro metric calculates string similarities based on the number and order of common characters. The second strong matcher is the attribute description matcher. It exploits natural language documentation snippets being available for elements in given schemas (e.g., the documentation of concept Cameras in Figure 2.20 could be “Cameras include all products from the photography domain, being it digital or analog ones. Camera accessories are not to be placed here.”). The TF-IDF measure [13] is adopted to compare the documentation of a schema element with all the other schema’s element documentations. For the example, the term “photography” in the description of Cameras would help to find the matching with Photography. Matchings that have been found by strong matchers are directly provided to the user. Remaining unmatched elements are processed by weak matchers. They use schema as well as instance data that is to be cleaned in advance for creating additional matchings. Two weak matchers have been implemented for ASID. The first one uses a Naive Bayes classifier and learns its decision model on provided instances. The second matcher is again based on TF-IDF. This time, documents are created consisting of all instances related to one schema element. These documents’ vectors can be compared with vectors of the other schema’s elements.

Combined matchers use schema information as well as instances for creating an alignment. This does not mean that such matchers are always superior to pure schema-based or instance-based matchers. Therefore, a comparing overview of all approaches will be given in the following and final schema matching section.

Comparison of the Approaches

A great amount of schema matching approaches has been developed in the last years. Only a small excerpt of related work could be presented in the previous sections. Many surveys and books on schema and ontology matching have tried to classify the different approaches using a multitude of features. In Table 2.4, three different features are used to categorize presented approaches. They include the information base (schema or instance), the element-level matching properties (string, language, data types, key properties, domains and ranges, etc.), and the information representation (ontologies, relational schemas, or XML). Structure-level properties are not taken into account since structure-level matchers vary widely between different approaches and are often quite specific for a domain.

It is to be seen that schema-based matching approaches have been developed at all times.
Table 2.4.: Comparison of Approaches for Schema and Ontology Matching.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Berlin and Motro</td>
<td>Auto-match</td>
<td>Attribute Dictionary</td>
<td>Instance</td>
<td>Naïve Bayes</td>
<td>Relational Schemas</td>
</tr>
<tr>
<td>2002</td>
<td>Do and Rahm</td>
<td>COMA (+++)</td>
<td>Flexible Matcher Combination</td>
<td>Schema</td>
<td>Strings, Lang., Types; Thesauri, Syn., Hyp., Abbr., Reuse</td>
<td>All</td>
</tr>
<tr>
<td>2002</td>
<td>Melnik et al.</td>
<td>Similarity Flooding</td>
<td>Flowing Matching Values</td>
<td>Schema</td>
<td>Strings, Types, Key Prop.</td>
<td>Relational Schemas, XML</td>
</tr>
<tr>
<td>2003</td>
<td>Doan et al.</td>
<td>GLUE</td>
<td>Joint Distrib. Probabilities</td>
<td>Instance</td>
<td>WHIRL, Naïve Bayes</td>
<td>All</td>
</tr>
<tr>
<td>2004</td>
<td>Euzenat and Valtchev</td>
<td>OLA</td>
<td>i-th Level Contributors</td>
<td>Schema, Instance</td>
<td>Strings, Lang., Types; WordNet</td>
<td>Ontologies</td>
</tr>
<tr>
<td>2004</td>
<td>Ehrig and Sure</td>
<td>NOM</td>
<td>Naïve Rules</td>
<td>Schema, Instance</td>
<td>Strings; Domain Vocabulary</td>
<td>Ontologies</td>
</tr>
<tr>
<td>2004</td>
<td>Ehrig and Staab</td>
<td>QOM</td>
<td>Candidate Selection Strategies</td>
<td>Schema, Instance</td>
<td>Strings; Domain Vocabulary</td>
<td>Ontologies</td>
</tr>
<tr>
<td>2008</td>
<td>Bozovic and Vassalos</td>
<td>ASID</td>
<td>Strong and weak Matchers</td>
<td>Schema, Instance</td>
<td>Jaro, Naïve Bayes, TF-IDF</td>
<td>Relational Schemas</td>
</tr>
<tr>
<td>2008</td>
<td>Duchateau et al.</td>
<td>Match-Planner</td>
<td>Decision Trees</td>
<td>Schema</td>
<td>Strings; Jaccard, Dictionary</td>
<td>All</td>
</tr>
<tr>
<td>2010</td>
<td>Akbari and Fathian</td>
<td>-</td>
<td>Neighborhood Matrices and Grids</td>
<td>Schema</td>
<td>Strings</td>
<td>Ontologies</td>
</tr>
</tbody>
</table>
This is clearly understandable since most of the integration tasks aim at the creation of correct alignments. Otherwise, the information in both input schemas is rendered unusable. Mixed systems using schemas as well as instances have only been developed recently. Furthermore, the amount of applied element-level matcher types seems to drop in newer matching applications. This is caused by the displaced focus in recent matching approaches. Previous works introduced a large amount of different element-level matchers exploiting miscellaneous properties of given schema elements. These matcher types do not have to be reinvented in more recent works and are often not mentioned in corresponding publications although being applied to given schemas. Instead, the focus currently lies more on how to choose and combine matchers automatically for a given task (Peukert et al., ASID, MatchPlanner, etc.). Finally, it can be seen that previous works were kind of omnipotent concerning the knowledge representations they could handle. This is only true in parts since these approaches mainly focused on taxonomic relationships or very general relations. Approaches only concentrating on ontologies take more ontology-specific features into account, such as special OWL constructs that cannot be reproduced in a standard database or XML dialect.

However, none of the currently available approaches for schema and ontology matching offers an integrated solution being generic and robust enough to build the basis for future developments. This caused Shvaiko et al. to publish ten basic challenges for the future development in ontology matching [158]. Those include (1) a more comprehensive evaluation method than the current OAEI benchmark, (2) a better performance with less usage of main memory, (3) discovering mechanisms for valuable matching background knowledge, (4) a better understanding of uncertainty in ontology matching, (5) effective matcher selection, combination, and tuning, (6) reasonable user involvement in the matching process, (7) explanation of the resulting alignment for the user, (8) enabling social and collaborative ontology matching, (9) establishing an infrastructure for alignment management, and (10) reasoning with alignments. Due to their high research potential, the matching routines to be developed in chapter 5 cannot satisfy all of these requirements (e.g., modeling uncertainty in ontology matching). Furthermore, the approach will focus on the product domain and thus cannot be called fully generic. However, according to (1), the system to be developed will be evaluated using a sufficiently large gold standard, (5) the matchers to be adopted will be tuned automatically to perfectly fit to the product domain, (6) the user will be involved in the matching process when feasible, (7) the resulting alignment will be explained in a traceable manner, (8) users will have the possibility to edit created alignments as well as the integrated matching ontology, and (9) previous alignments will be reused.

When talking about extracted product specifications, structural information is not available, thus reducing the set of elementary matchers to the element-level ones. For compensating this major drawback, as many characteristics of product specifications as possible have to be identified to be exploited by adequate element-level matchers. The wide matcher set available through COMA will be used as an inspiration here. Furthermore, the fact that not only the specifications’ schema (TBox), but also corresponding instances (ABox) are extracted, puts a focus on the domain of combined matching and corresponding approaches. Combined matchers and instance-based matchers often use machine learning
techniques which might therefore also be interesting for this work.

In any case, a comprehensive view on the product information domain is required before actual techniques can be developed to cover this domain. Hence, it will be inspected in detail throughout section 2.5 after a short overview of information presentation.

2.4. Information Presentation

Information presentation is the last step to be performed in a federated information system. While document retrieval, information extraction, and information integration clearly remain in the backend of such a system, information presentation includes all technologies of a federated information system’s frontend. Presenting information in an appealing way is of utmost importance since a user may cease to utilize a system immediately if it does not provide her the demanded information in an expected manner.

However, since the focus of this research project does not lie in the area of frontend technologies, related work will not be considered here. For enabling effective product comparisons, section 6.1 in the evaluation chapter will still provide some details on how information is presented to users and how it is made available for reuse in other applications.

The previous sections presented federated information systems in general. Since the approaches to be developed in following chapters will especially focus on federated product information search, the next section will discuss the applied terminology of the product information domain.

2.5. Product Information

Product information is a rather general term. It is often used in the context of product information management [183] (PIM) which refers to the central organisation of information about products, especially focusing on marketing and selling such products. During this work product information shall include all kinds of information about products available, regardless of its purpose.

As the usage of a clear terminology is important for this work, more detailed descriptions are needed for product information source characteristics, product information source types, product information integration types, and product information types. These will be given in the following.

2.5.1. Product Information Source Characteristics

In chapter 1 three types of product information sources have been mentioned. However, two different sources belonging to the same class might not be as homogeneous as it seems. For example, a vendor source like Amazon being available through a Web Service has to be handled differently from Buy.com which only offers a very basic service and thus has to be accessed through its web interface. In fact, the information source provider, from which the categorization originates, is just one of many characteristics that can be assigned to
a product information source. Concerning this work, the provider is the most important
criterion and thus a division by this property is legitimate. Still, other properties are of
relevance as well since they constitute the requirements a federated product information
system has to fulfill for integrating them. All relevant source characteristics will be
presented in the following to complete the picture of a product information source. The
criteria have also been devised in [150].

**Product Information Source Provider.** Product information can be supplied by different
types of providers. Rather than dividing provider types by instances like concrete
companies (e.g., Amazon, Nikon [130], or Audi [3]), the division is made on the
concept level. That is, the three already defined provider types vendor, producer, and
third party are adopted.

**Product Information Source Dynamics.** Information sources, and thus the information
they provide, can underlie different dynamics. These dynamics are influenced by a
series of factors like a web page being static or dynamic, the number of authors
working on a source, the web page belonging to a news website, etc. Accordingly,
product information sources can be static, dynamic, or high-grad dynamic.

**Product Information Source Medium.** The media provided by a product information
source heavily influence the way how information is to be handled by a federated
search system. If the representation is textually, included information can be
extracted and processed. Image representations are more complex to process. If
they contain text snippets, techniques like Screen Scraping may help to extract
some information. The task is getting even harder for video files. The media types
available can be split into text, image, video, audio, and interactive (e.g., Adobe
Flash).

**Product Information Source Format.** The format of the information in a product infor-
mation source describes the serialization that is used to represent the product
information. If talking about text media, information being represented as standard
text can be accessed easily. Product information that is wrapped in binary files is
harder to access as a dedicated API is required. Additional problems emerge if the
binary format is proprietary, thus baring a programmer from creating an API to
access that format. According to the mentioned details, information source formats
are to be divided in text, binary free, and binary proprietary.

**Product Information Source Location.** The location of an information source influences
the disposability of product information as well as the time needed to retrieve it from
the addressed source. If the source is not operated by the owner of the federated
information system, important information may be unavailable temporarily. If
the source is running on a machine with a slow internet connection, it may take
too much time to retrieve information. According to these statements, product
information source locations are divided into local and remote.

**Product Information Location Notice.** The Web offers a large number of product infor-
mation sources. An ideal federated information system would integrate all sources
Product Information Source Structure. Information sources can deliver product details of varying structures. The different levels of this characteristic can be divided into structured, semi-structured, and unstructured. Structured sources follow a distinct schema that can be analyzed in advance, thus easing the information processing heavily. Semi-structured sources are sources only containing some basic structure which is not known in detail. Finally, unstructured sources do not contain any data schema (continuous text). This distinction can be made on the inner or the outer structure. The inner structure is document type-independent and determined for each information source or document separately. For example, a text file may be structured if its content consists of comma-separated values. On the other hand, a database table would be categorized as unstructured if it contains comprehensive product descriptions. The outer structure can be determined without examining the contents. It only depends on the source or document type providing the information. Thus, structured sources are databases or XML documents with an XML Schema Definition (XSD). Web pages are typically categorized as to be semi-structured since they are written in HTML. HTML provides some basic idea about the title of a document or information contained in a table. Eventually, unstructured sources are other document formats like PDF or ODT which are mostly to be processed by the use of Natural Language Processing. Although the inner structure of a source or document is more meaningful, the outer structure is generally used to classify documents and sources (see also Figure 2.8). Thus, it will be used throughout this work as well.

Product Information Source Access Protocol. The access protocol of a product information source describes the protocol in terms of the Open Systems Interconnection (OSI) reference model. Generally, only the three highest layers (transport, application, and presentation layer) of the ISO/OSI [64] model are of relevance. Access protocols include, e.g., HTTP or FTP.

A product information source can be fully specified using the aforementioned source characteristics. To illustrate the usage of those characteristics, the Amazon web page has been specified with concrete values in Figure 2.23.

As algorithms to be integrated in a federated product information system are always adapted to the specifics of the information sources to be included, it is of high importance to particularize these sources before designing the actual algorithms. The following section will take care of this task.
2.5.2. Product Information Source Types

Each of the previously mentioned product source characteristics is important for being able to design adequate information search and integration algorithms. As already depicted, the main characteristic for categorizing product information sources will be the provider. In the following, each of the three resulting source types is defined explicitly.

**Vendor Source.** A vendor source is an online shopping mall offering products from at least one producer that is not the vendor itself. Typically, the available product set is located in different categories while specialization on a few categories is not unusual. The information available per product in a vendor source can be of varying quantity. It contains details such as the product’s name (e.g., “SLR 38”), its producer’s name (e.g., “Digi”), a price (e.g., “$1,599”), or a picture. Depending on the vendor, additional information like a description may be available. As the products are not normally produced by the vendor, most of those products can be sold by other vendors as well. Vendor sites offer a search form for accessing their product catalog and retrieve results pages with found products. Each product can then be examined in a detail view. Concerning the automated interaction with information sources, some vendor sources maintain Web Services for accessing their product catalogs, thus delivering structured product information. If no Web
Service is available, the portal can only be accessed directly using its website, thus providing semi-structured information. As the vendor itself is not of high interest but rather its online representation, the term “vendor” will be used synonymously with “vendor source”. Examples for vendors are Amazon, Buy.com and, eBay.

**Producer Source.** A producer source offers a website for a company that assembles products. Producers specialize on few or even only one product category. As the products presented on the online portal are produced by the presenting company, no other producer source offers details for those products. Typically, producers present their products using a product detail page consisting of description texts (e.g., “The D60 is a digital high-performance single-lense reflex camera...”), advertisement (e.g., “This camera will change your life.”), and the product’s specifications (e.g. “Effective Pixels: 10.2 million”). Sometimes many products are presented jointly on one page. Producer information can only be accessed using the producer’s website as the maintenance of Web Services does not pay off in this case. Thus, information is either semi-structured (product specifications) or unstructured (description texts).

Since the producers themselves are not of high importance for this work, the term “producer” will be used synonymously with “producer source”. Examples for producers are Nikon, Audi, and Sony [163].

**Third-Party Source.** A third-party source offers a website that presents product information and is not a vendor or producer. This includes different types of online product information sources. In many cases, third parties present user-generated content. The term *user* includes average consumers offering their product experiences in product forums or blogs (e.g. “I bought this camera and I am really excited about its excellent quality.”). Alternatively, third parties may offer professional content for different product categories. Such content is to be found on test websites or professional blogs (e.g. “When you open up the camera, you see that Digi did a clean job on positioning the different components.”). The information given in third-party sources is generally unstructured as it consists of full text. Similarly to producers, Web Services are generally not provided by third parties. The term “third party” and “third-party source” will be used synonymously for the same reasons as mentioned above. Examples for third parties are Digital Photography Review, Ciao!, and Edmunds.com [6].

As not all characteristics of each source have been emphasized in the definitions above, Table 2.5 shows the three categories again, this time including the specification of all other product source characteristics as well.

The challenge for a federated product information portal is to integrate all presented information source types to be able to generate a product picture that follows the criteria defined in section 1.4.1. During integration, the source peculiarities listed above have to be taken into account. Different algorithms are to be developed for this purpose. The goal of such algorithms is to convert the product information to a general format that is identical for all products in all domains. The next section pictures details on the different levels of information integration that have to be distinguished.
Table 2.5.: Source Characteristics of Vendors, Producers, and Third Parties.

<table>
<thead>
<tr>
<th>Provider</th>
<th>Vendor</th>
<th>Producer</th>
<th>Third Party</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamics</td>
<td>(High-Grad) Dynamic</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Medium</td>
<td>All</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>Format</td>
<td>Text or Binary Free</td>
<td>Text or Binary Free</td>
<td>Text or Binary Free</td>
</tr>
<tr>
<td>Location</td>
<td>Remote</td>
<td>Remote</td>
<td>Remote</td>
</tr>
<tr>
<td>Location Notice</td>
<td>Known</td>
<td>Unknown</td>
<td>Known or Unknown</td>
</tr>
<tr>
<td>Structure</td>
<td>(Semi-)Structured</td>
<td>Semi-Structured</td>
<td>Semi-Structured</td>
</tr>
<tr>
<td>Access Protocol</td>
<td>HTTP</td>
<td>HTTP or FTP</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

2.5.3. Product Information Integration Types

As mentioned above, the most important product information source characteristic is the one of the information source provider. Furthermore, the structure of an information source is relevant since it influences the type of adopted extraction algorithms heavily. Based on these two characteristics, the integration of product information can be separated into three types, i.e., locating product information, syntactic integration, and semantic integration. These types may equally be mapped to information access, extraction, and integration. Figure 2.24 shows the resulting cube taxonomy for product information search and integration on the WWW.

This taxonomy was developed for presenting the different dimensions in product information integration and offers great help in understanding emerging problems of federated product information systems. Different gray shades show the complexity of the individual tasks. Light grey fields imply that the corresponding task can easily be executed using state-of-the-art technologies. Normal grey fields indicate a higher degree of complexity as a whole bunch of ideas and algorithms is required to solve the emerging problems. Tasks marked with dark grey are of the highest complexity. They are hard to solve and possible solutions might return unfeasible results. Lastly, the six missing cubes in the front signify that these tasks do not make sense since producer and third-party sources generally do not provide Web Services or comparable mechanisms.

Every field is completed with a short information snippet giving an idea of what is meant by the combination of the two corresponding dimension values. In the following, the goals and non-goals of this work already presented in the introduction will be revisited using the dimensions given in the cube taxonomy. The shortcuts will be Ven/Pro/TP, Un/Semi/Struc, and Loc/Syn/Sem for the information source provider, the information source structure, and the information integration type, respectively.

Considering vendor information, only structured and semi-structured (Ven-Struc and Ven-Semi) sources are to be handled. Furthermore, the integration level will only be syntactic (Ven-Syn), that is, locating and semantically integrating vendor information...
Figure 2.24.: Cube Taxonomy for Product Information Search and Integration.

has to be done by the user. Vendor information is important for gathering bootstrapping information, thus, developed techniques are only a by-product of this work.

The main focus will lie on processing semi-structured producer information (Pro-Semi). Here, all steps for locating information sources, extracting relevant information, and semantically integrating it are to be executed.

Finally, third-party information (TP) is to be examined. It will only include some basic approaches for finding and extracting such data (TP-Un-Loc and TP-Un-Syn).

All product information which is retrieved from different sources needs to be managed using a general representation. This representation is described in the next section.

2.5.4. Product Information Types

Product information can be available in different forms. It may consist of details such as a product name, a price, or a picture. Additionally, describing texts or user opinions may
be valuable. Such continuous texts may be broken down to atomic information such as the aforementioned product name. However, the final product information needs to be presented in a dedicated data structure. Additionally, a clear terminology is important for the subsequent descriptions. Therefore, the following definitions will describe how product information is represented.

**Product Specification.** A product specification is defined as the pair of a product attribute and its corresponding value. This might be a product’s name, a product’s price, or the available resolution if the product is a digital camera - each including the corresponding value. Thus, a specification is a relation like: "name" => "Digi SLR 38".

**Product Specification Set.** Multiple product specifications are called a product specification set. A product specification set is a map, e.g., {"name" => "Digi SLR 38", "resolution" => "10.75 MP", ...}.

**Product.** If a product specification set comprises sufficient specifications for one product, it is equal to the term product as the good itself is not treated by a federated information system. Rather, only the information about a product is what the system can deal with. Hence, a product is of the same data structure as a product specification set.

**Offer.** Products may be provided by different vendors, each of them describing the product a little different, e.g., by assigning another price. Thus, an offer consists of a product and additional vendor-specific details, e.g., {"name" => "Digi SLR 38", "price" => "$599.99", "resolution" => "10.75 MP", ...}.

**Product Set.** A product set is a collection of products. According to the previous definitions this is equal to a collection of specification sets. Thus, a product set is a collection of maps, e.g., [{"name" => "Digi SLR 38", ...},{"name" => "Digi Camcorder 909", ...}, ...]. If each product specification set contains a unique key that is to be used as an ID, the set is sortable and thus changes the data structure to a product list. Although presumably all vendors use unique IDs to identify their products, the term product set will be preferred over product list in this work as the id is often unaccessible from outside the vendor’s network.

**Product Portfolio.** All products sold by a specific vendor are part of this vendor’s product portfolio. Thus, a product portfolio is similar to a product set, being distinguished by the fact that it contains all products of the corresponding vendor.

Having a clean terminology for the product information domain as well as a detailed description of the information sources to be included at hand, corresponding algorithms for integrating such information sources can be developed in the next chapter. The final section of this chapter will provide some conclusions on federated information systems and federated product information systems.
2.6. Conclusions

This chapter focused on the basics of federated information systems in a web context. All relevant steps for such systems, including information access, information extraction, information integration, and information presentation have been described. The different sections are based on general developments in each of the according research directions as well as approaches trying to find a solution for one specific problem of the corresponding research area.

Information access has been treated on a higher level including document retrieval, federated search, and federated ranking. Presented approaches offer ideas on how to integrate information from different sources in one knowledge repository. However, none of them is specifically suited for the problem tackled in this work, namely, the directed identification of valuable product information sources and the effective ranking of online malls.

The information extraction section, especially the area of IE from semi-structured sources does offer algorithms potentially helpful for this work. Some related approaches have been identified to build the basis of required techniques. However, since the product domain includes various peculiarities regarding the information extraction, more conceptual work is demanded.

The information integration section focused on schema and, specifically, ontology matching. Extracted product specifications need to be integrated with a common terminology, e.g., being represented by an ontology. Especially element-level matching is helpful in this case and described matchers will be adopted in a composite matcher being applicable in the product domain.

Information presentation has not been tackled since it is to be seen as a side product here. The last section included an all-embracing definition of terms being relevant for the product information domain. Based on this terminology and the examined technologies and approaches for federated information systems, the next chapter offers a short preview on the federated product information system to be designed before the main conceptual chapters step into details of the corresponding system and its implementation called Fedseeko.
A Federated Product Information System

The previous chapter introduced the general idea of federated information systems. The classical data flow in such a system consists of information being accessed through different protocols or interfaces, the filtering of this information through information extraction techniques, the integration of information, and finally an adequate information presentation. Naturally, the details of this data flow depend on the type of available information, the domain the system is working on, the technologies to be adopted, etc. Accordingly, the federated architectures presented in the last chapter differed heavily from each other.

The data flow of this work’s product information system needs to be adapted as well. Since the focus is on product information from the Web, the idea is to access online malls for retrieving relevant offers. After having identified the basic product specification sets in these offers, they can be enriched by additional product specifications from producer sites. For uniformity reasons these specifications need to be integrated with each other. The resulting complete product specification sets may finally be presented. The complete data flow is depicted in Figure 3.1.

![Figure 3.1: The FEAD Chain - Find, Enrich, Administrate, and Display Information.](image)

As suggested by the figure’s caption, the different research fields can be classified into a series of steps belonging to the FEAD Chain. Each step consumes (“feeds on”) the results
of its previous step and produces an extended result for the following one. The first step, that is, finding ("F") basic product information to be used as bootstrapping information, requires the application of federated search and federated ranking techniques. This is due to the adoption of online malls for gathering basic product information sets. The second step enriches ("E") this basic information with technical product specifications. Thus, some sort of document retrieval and information extraction is required. Furthermore, gathered product information is to be managed or administrated ("A") in step three. Here, some data model is needed to manage such information which may be represented by ontologies. Additionally, gathered information needs to be matched with this data model (ontology matching). Finally, the information is to be displayed ("D") in step four to allow product comparisons where techniques like faceted search and semantic comparisons are of importance. The overall process has been published incrementally in [151], [179], and [168]. Each step will be provided with some more details in the following.

3.1. Finding Basic Product Information

Retrieving offers from vendors includes querying their online malls through a Web Service or web portal, potentially extracting retrieved results from different results pages, integrating the results with the ones of other vendors, and ranking them depending on some defined criteria. This application flow is quite similar to the one of the IPIS system [111, 110]. IPIS unfortunately relies on available Web Services and expects them to handle complex query configurations which makes it a quite limited approach. The intended architecture should be able to retrieve basic product specifications from simple web pages as well.

The amount of product specifications being available through a vendor’s Web Service may be quite comprehensive. However, since most vendors only offer web portals with human-readable content and an inspection of each result’s detail page would be too costly in terms of time, the information to be retrieved per product has to be constrained to the maximum set of specifications being available on almost every vendor results page. An example for such information is given in Figure 3.2.

```java
product = {
    "name" => "SLR 38", "producer name" => "Digi",
    "picture URL" => "http://img.digi.com/slr/slr38.png",
    "description" => "The SLR 38 is a new camera made by Digi."
}
offer = {
    "product" => product, "price" => "$1,599",
    "detail_page_url" => "http://slr-shop.com/products/digi-slr38.html"
}
```

Figure 3.2.: Offer Information from Vendor Sources.
When investigating vendor results pages, six fields shape up as this greatest common specifications set. They include four product-specific details, namely, the product’s name, the product’s producer name, a picture URL, and a description, as well as two offer-specific details, namely, the offer’s price and its detail page URL. These details may either be retrieved at query time or by a crawler, permanently discovering new offers and adding them to a database. In the case of query-time extraction, the user’s query needs to be preprocessed in order to create a reasonably sorted list of offers being retrieved from different vendors.

The basic product information set is to be extended in the next step.

### 3.2. Enriching Product Information

As described in the previous section, product offers coming from different vendor sources cannot be expected to contain many product specifications. Anyhow, especially for electronic products, technical product specifications are of topmost importance to a potential consumer. Thus, such specifications need to be identified on the Web using search engine-supported crawling mechanisms. When an adequate page, preferably being located on the product’s producer domain, has been found, extraction mechanisms are to be adopted to filter out the actual product specifications (e.g., the resolution of a digital camera) and to enhance the initial shallow set of specifications. The identification of such web pages is based on the product specification set taken from a retrieved offer. The extraction may use labels that are provided by a user or knowledge from previous extractions in the same product domain. Again, the information enrichment may be initiated by a consumer searching for some special product or by a crawler. The resulting data structure may look like in Figure 3.3 (producer name, picture URL, and description are not shown).

```plaintext
product = {
    "name" => "SLR 38", ..,
    "total pixels" => "10.75 MP", "effective pixels" => "10.2 MP",
    "height" => "15 cm", "length" => "20 cm"
}
```

Figure 3.3.: Product Information from Producer Sources.

The idea of enriching the product information base by additional structured information automatically has also been adopted in CrIP [145] and Aletheia [181]. Both systems include concrete extraction services for semantically indexing documents. Although being provided with semi-structured documents, they do not focus on detecting content structures and only apply techniques of unstructured information extraction. Furthermore, the document indexing is driven by a previously modeled domain ontology. The extraction component of the FEAD architecture in contrast will mainly be based on document structures and merely use domain knowledge as supporting extraction hints. For this
task, ideas of presented approaches on semi-structured information extraction will be helpful.

After having augmented the product specification sets with corresponding technical specifications from producer pages, the integration step can be executed.

### 3.3. Administrating Product Information

Extracted product specifications vary depending on the domain the product is located in and their provider. The aim of integrating such specifications is to always denote product specifications of one category with the same terminology. Thus, some kind of knowledge representation for supported product categories is required. Ontologies are an appropriate instrument for modeling product information by defining product categories, relations between those categories as well as available product specification types for each category. Ontologies like GoodRelations [91, 92] or eClassOWL [88] have been specifically developed for the product domain. Unfortunately, GoodRelations does not model technical product specifications. eClassOWL provides only a very limited and therefore insufficient set of such product details. Hence, an adequate ontology needs to be created for this task. Having such an ontology at hand, extracted product specifications are to be matched with the available ontology concepts. Several techniques of the previously presented schema matching approaches can be adopted here. The resulting homogeneous product specification sets may then be provided to other applications through a machine-readable format in order to further process them. Depending on the ontology, such homogeneous information might look like in Figure 3.4 (again, basic product specifications have been left out).

```json
product = {
    "name" => "SLR 38", ...
    "resolution" => {"total" => "10.75 Megapixels",
        "effective" => "10.2 Megapixels"},
    "size" => {"height" => "5.9 inches", "length" => "7.9 inches"}
}
```

Figure 3.4.: Product Information in a Clean Format.

An application to further process homogeneous sets of product specifications may display the information for potential consumers as described in the following.

### 3.4. Displaying Product Information

Being provided with a set of products, each consisting of a set of product specifications, an application could present this information to a potential consumer in an appropriate manner. Technical specifications may be clustered and offered as facets in a web interface.
Using these facets, the vast amount of products can be reduced to a small group of interesting products which may be compared like shown in Figure 3.5.

<table>
<thead>
<tr>
<th>Digi SLR 27</th>
<th>Digi SLR 38</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resolution</strong>:</td>
<td></td>
</tr>
<tr>
<td>Resolution &gt; Total:</td>
<td>8.5 Megapixels</td>
</tr>
<tr>
<td>Resolution &gt; Effective:</td>
<td>8 Megapixels</td>
</tr>
<tr>
<td><strong>Size</strong>:</td>
<td></td>
</tr>
<tr>
<td>Size &gt; Height:</td>
<td>4.5 inches</td>
</tr>
<tr>
<td>Size &gt; Length:</td>
<td>5.9 inches</td>
</tr>
</tbody>
</table>

Figure 3.5.: Presentation of Product Information.

Finally, the homogeneous representation of the products’ specifications may allow semantic product comparisons, e.g., if the application points out that one camera’s effective image resolution is two times as big as another one’s.

3.5. Conclusions

In this chapter, the data flow of a federated product information system has been presented. The basic steps consist of finding bootstrapping information for products in online malls, extending this information with a producer’s list of technical product specifications, bringing the available information into a homogeneous format, and presenting the information to the user in an appealing way.

In the following, the different algorithms for creating such a homogeneous product data set will be conceptualized. The concept is divided into two chapters, the first of which describes the information extraction parts (chapter 4). It includes the retrieval of information from vendors, producers, and third parties. The second concept chapter (chapter 5) focuses on information integration, that is, ontology matching. Finally, the results are evaluated based on an implementation called Fedseeko in chapter 6.
Product information extraction constitutes the first of two central parts of the presented concept. It covers the handling of all information provider types, each being analyzed in a subsection. The first section (section 4.1) presents algorithms on vendor information search. Techniques for querying vendors and extracting their retrieved product information are described before the ranking of such information allocated by different vendors is demonstrated. The second section (section 4.2) describes how to retrieve a product page based on given offer sets and the process for extracting product specifications out of these pages. Since the theses given in the introduction chapter mainly focus on product specifications retrieved from producer pages, this section is the most important one of the extraction chapter. Finally, basic information on the inclusion of third-party information is given (section 4.3). A conclusion completes the chapter (section 4.4).

All sections will be using pseudo code to give an idea of how the algorithms may be implemented. The syntax is close to the Ruby programming language. Appendix A.1 presents some additional information on how to read the code.

Figure 4.1.: The FEAD Chain - Find Basic Product Information.
4.1. Vendor Product Information Search

The sheer number of online shops and their heterogeneous ways of product presentation as well as differing quality standards complicate the process of searching relevant offers for the average Internet user and create an annoying and longsome experience. For being able to avoid this situation, algorithms need to be developed that allow querying several vendors simultaneously through a unique query interface and efficiently ranking their offer search results at query time. Such algorithms are located in the finding step of the previously introduced FEAD Chain (Figure 4.1).

In the related work section about federated product information search (section 2.1.2), different systems have been presented, each of which had some serious drawbacks. Especially IPIS \[111, 110\], the most similar system to the one to be developed here, only offered the connection of Web Services for retrieving product information and did not present a practical solution for the offer ranking problem. Thus, a new architecture has been developed, the conceptual data flow of which is presented in Figure 4.2.

![Figure 4.2.: Vendor Product Information Search.](image)

As can be seen in the figure, generating uniform human- or machine-readable product offers requires a series of different steps. Initially, the process is provided with a query and the vendors that are to be taken into account. Then, the query needs to be categorized (steps 1 to 4) for being able to decide which of the provided vendors are relevant concerning the current query. The vendors have been rated for each category in advance. Using this category, an overall ranking is calculated (step 5) and relevant product offers are retrieved for the query (steps 6 to 9). The offers may either be taken from a Web Service (steps 6a to 9a) or, if no Web Service is available, extracted from returned HTML pages.
Vendor Product Information Search

(steps 6b to 9b). The offers are assembled according to the precalculated ranking and finally returned to the querying instance (step 10).

As already stated in section 1.4.2, vendor product information search is not in the main focus of this work. For uniformity reasons, a set of requirements derived from the process described above will still be provided here. The requirement identifiers are numbered with a prefixed zero since vendor information search takes place before producer product information search.

**Req 0.1** The system needs to be able to rate a vendor concerning its suitability for a given category.

**Req 0.2** A query that is provided by a user or accessing application needs to be categorized in order to decide which vendors are relevant for that query.

**Req 0.3** Product offers need to be ranked and retrieved from different vendors, independent of the format a vendor chooses to provide the offer details.

The following sections present algorithms for satisfying each of the requirements, being separated into a vendor product information ranking and a vendor product information extraction part. Intermediate results of this section have been published in [178] (see also [108, 56]).

### 4.1.1. Vendor Product Information Ranking

Good ranking and filtering strategies are essential for federated product search. Unfortunately, federated ranking mechanisms are hard to realize since there is no objective measurement such as the link structure for web pages available. Neither structure-based mechanisms (e.g., PageRank), nor content-based ranking approaches (e.g., TF-IDF or BM25 [106]) are applicable to solve this problem. The situation is even worse if a provided query does not specify a concrete product, but consists of general terms such as “digital camera” which disables a federated product search system from recognizing a distinct product and ranking results for this product higher than, for example, available add-ons. Additionally, federated search providers generally do not have information about sales numbers or customer satisfaction values for products or shops at their disposal.

In this section, a category-based ranking algorithm is presented that tries to mimic the way a user with expert knowledge in the current product domain would select shops and order product offers from these shops. The developed method executes the federation and ranking mechanisms at query time. This is a basic requirement since the majority of online malls cannot be crawled in advance.

The idea is to first map a given query to a product category (e.g., “Digital Cameras”) and rank the chosen shops according to their competence in the detected category. Some shops may be specialists for digital cameras, others for books, or may have a good reputation as general-purpose stores. Only the best-ranked shops actually receive the query and return individual result lists. An automatically generated product ranking then has to correspond to an ordered list created by a human being having expert knowledge in the respective domain when assigned to the same task. Of course the automated
approach should complete in less time and with a larger amount of information taken into account. The conditions shown in Table 4.1 should also be fulfilled by the algorithm.

Table 4.1.: Requirements for a Federated Ranking Algorithm.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traceability</td>
<td>A consumer may retrace why an offer appears higher in the results list than another one.</td>
</tr>
<tr>
<td>Balance</td>
<td>Scores are calculated by overall relevance, not just single features.</td>
</tr>
<tr>
<td>Equal Treatment</td>
<td>Single sources or products are not preferred in comparison to others.</td>
</tr>
<tr>
<td>Reproducibility</td>
<td>Results are always scored the same way resulting in the same outcome.</td>
</tr>
<tr>
<td>Scalability</td>
<td>Scores are easy to compute to enable a query-time solution.</td>
</tr>
<tr>
<td>Completeness</td>
<td>All available sources are included.</td>
</tr>
</tbody>
</table>

Since there are no major standards for the design of a method to rank product search results obtained from different online malls, this method shall be inspired by the way human beings perform federated shopping tasks on the Web. The model to be used throughout this section is called “The Federated Shopper” and is presented in the following.

1. Gain experience which shops are best qualified for particular product categories (permanent process).
2. Detect the category of the product to be searched.
3. Choose shops that are suitable for the corresponding category.
4. Query shops for the product and create a ranked result list taking the shops’ internal rankings as well as their suitability for the current category into account.

The algorithm is based on estimated scores that are assigned to each offer depending on the relevance of the corresponding vendor for the offer’s category as well as the vendor’s internal ranking for the considered offer. Each vendor’s relevance scores for the available categories need to be calculated initially. Therefore, several sample lists of products from different categories have to be created. Then, for every tied vendor, it needs to be checked which fraction of these products is contained in the underlying database. This idea is similar to the one presented by Si and Callan [159]. The estimated vendor score for the product’s category is merged with the vendor’s internal ranking score of each product to calculate a total ranking for all products. Thereby, a result set retains the ranking order of the vendors, but tends to rank products of vendors higher which are more relevant for the current category. The respective steps are explained in-depth below.
Product Samples Collection

To evaluate each vendor’s relevance for a certain product category, a sampling process has to be executed with a set of product titles. Obviously, it is necessary that the product lists used for sampling each source clearly shape the market segments. Unbalanced sets may easily lead to overspecialization in certain areas (e.g., a sample set for “Books” only containing titles on sports) causing incorrect relevance estimations.

Different methods to compose the sample sets are available. The most evident approach creates the sets using expert knowledge. This is a very time-consuming task but certainly shows good results in evaluation. Alternatively, an algorithm could be used to randomly query several shopping portals based on keyword lists. The automatic creation of such lists being representative for every domain would involve additional research work.

Eventually, the favored method gathers products by crawling available categories of big shopping portals like Amazon. For every category, the crawler selects a random page, followed by the selection of a random product and the extraction of its title. This process continues until a suitable set has been created. As the employment of only one big vendor for the category crawling might produce biased product sets, the product samples collection routine relies on the usage of several online malls.

Vendor Ranking

As mentioned before, each vendor is assigned a relevance score for each product category by sampling it with the corresponding product set (Figure 4.3). This process has to be done initially once for every vendor and may be repeated to keep the scores up-to-date.

When being sampled, every vendor returns the total number of hits for each category which could already be viewed as an estimation of a vendor’s relevance concerning a category. However, it does not fully reflect the model of the Federated Shopper since shops with smaller portfolios should still be preferred over others if they are specialized on certain categories. The formulas for vendor ranking take this fact into account.
As shopping portals generally do not provide information about the number of products contained in their catalog, the set of products available from a source src is defined as the sum of hits for all queries in all categories sent to this source (Formula 1.1).

\[
products(src) = \sum_{t=1}^{n} hits(cat_t, src) \quad (1.1)
\]

The quotient of the products amount found for a category cat and the number of products available in the source src results in the relevance of src for cat (Formula 1.2).

\[
relevance(cat, src) = \frac{|hits(cat, src)|}{products(src)} \quad (1.2)
\]

Dividing the hits count by the total number of products estimates the degree of specialization of a source’s catalog for the current category. This causes highly specialized stores to be scored better than stores covering many different categories. An idea of how the defined ranking functions can be implemented is given in Algorithm 4.1.

**Algorithm 4.1 Calculating Category Scores for Vendors.**

```ruby
1 vendors = get_all_vendors
2 category_samples = {
3   Category("name" => "Digital Cameras") => ["D60", "Finepix s1500", ...], ...
4 }

5 # Iterate through all categories with their samples and calculate the
6 # corresponding category scores.
7 category_scores = []
8 vendors.each do |vendor|
9   vendor_offers_count = 0
10  category_samples.each do |category, products|
11     category_offers_count = 0
12     products.each do |product|
13       category_offers_count += vendor.get_offers_count(product)
14     end
15     category_scores << {
16       "vendor" => vendor,
17       "category" => category,
18       "score" => category_offers_count
19     }
20   vendor_offers_count += category_offers_count
21 end
22 category_scores.map do |category_score|
23   category_score["score"] = category_score["score"]/vendor_offers_count
24 end
25 end
26 return category_scores
```
In the first line, available vendors are retrieved. A data structure containing categories and respective product samples like the one in lines 2-4 is provided by the samples collecting component. The algorithm itself iterates through the vendors and category samples, retrieves the available offer counts for each product (line 14), creates initial scores for each vendor-category pair (lines 16-20), and finally divides all scores by the overall amount of available products per vendor (lines 23-25). The resulting data structure contains map structures, each consisting of a vendor, a category, and a dedicated score describing the suitability of a vendor for a certain category. For example, the Amazon Product Advertising API might get a score of 0.947 for the “Digital Cameras” category while online shops like Otto might perform quite bad in this domain. Amazon results for queries categorized as to belong to the digital cameras domain would thus be ranked better than Otto results.

With the help of the categorization functionality presented below, search results from different vendors can be ranked.

Query Categorization

The ranking approach to be presented in the follow-up section needs to know the category a provided query belongs to, so that it can calculate the correct relevance estimations for potential offers and pick the appropriate sources to forward the query to. Since there are already large sets of classified product data available on the Web represented by online shops and portals that offer products and sort them into categories, this algorithm categorizes a query by passing it to these sources, using a majority vote on the actual product category. For example, Amazon and eBay could be queried for “digital slr”, their category names would be matched with internal categories (e.g., “Digital Cameras”) and then the category with the most hits for the current query would be picked. The matching of vendor site category names to the internal categories is done by using the Levenshtein distance as well as pattern matching or by the adoption of knowledge sources like WordNet.

As the classification of a single query may require multiple requests to vendor services, a concept for caching classifications and offline inferences of a query’s category enhances the approach. For each query passed to the system, the algorithm remembers the query’s substrings as well as the allocated internal category. When a new query is received, categorized substrings of that query are retrieved. In case the amount of previously classified substrings exceeds a certain threshold, the category can be inferred without external help. This method takes advantage of the similar naming within product lines and drastically reduces requests to vendor services after an initial training period. The offer ranking algorithm presented in the following makes use of the detected category.

Offer Ranking

Having a set of vendors with their suitability scores as well as a categorization functionality at hand, the actual scoring and ranking of product entries in a result set can be calculated (Figure 4.4). First, the provided query (e.g., “digital slr”) is classified. Therefore, the
method explained in the previous section to assign a category to a query is used (in this case, “Digital Cameras”). By means of this information, vendors are selected that are expected to return the best results by calculating the ranking values of the product search hits they would return (in the figure, Amazon and Otto).

According to the ranking values, some of the chosen vendors are queried for one or more results pages (e.g., the first and the second results page of Amazon as well as the first one of Otto). Finally, results are merged using the previously calculated ranking values. The formula for scores estimation is shown below (Formula 1.3).

$$\text{score}(hit, q, \text{cat}, \text{src}) = \text{relevance}(\text{cat}, \text{src}) \cdot \frac{|\text{hits}(q, \text{src})| - \text{position}(hit, q, \text{src})}{|\text{hits}(q, \text{src})|}$$ (1.3)

The score is made up of three components: the initially calculated relevance of a source src for the entry’s category cat (e.g., 0.947 for Amazon in the category “Digital Cameras”), the score that src originally assigned to the returned hit which is assumed to be equal to the entry’s position in the source’s results list (e.g., 0 or 1), and the amount of returned hits for the query q and source src (e.g., 52). As the figure shows, Amazon’s first results would be scored by $0.947 \times \frac{52-0}{52} = 0.947$ while the second result received $0.947 \times \frac{52-1}{52} = 0.929$. Therefore, product search hits originating from different vendors are arranged corresponding to the sources’ relevance estimations while preserving the order of entries from each vendor, respectively.

With the described ranking algorithm, the most important requirement, namely, the calculation of ranking scores at query time, can be fulfilled since it accounts for the fact that intensive results evaluation might cause serious performance losses (Paltoglou et al. [138]). However, the actual offers returned by the online shops are not known in advance. The calculated ranks are therefore independent from these offers. A drawback arising from this situation is that several returned offers from different vendors could reference the same product. Potential consumers might though not judge this fact as to be adverse because it gives them the possibility of comparing prices.
Furthermore, the described process has the edge over Si and Callan’s approach in the sense that no centralized ranking algorithm is required for deciding about a source’s quality. As mentioned in section 2.1.3, such an algorithm could hardly be judged as a neutral authority.

Having a categorized query as well as the calculated relevance scores for potential offer search results from different vendors at hand, these results can be filled with concrete offers from the most relevant vendors. This process is detailed in the following sections.

4.1.2. Vendor Product Information Extraction

Some vendors present their product information through Web Services. Web Services offer a comfortable means for making information accessible over computer networks. The provided data may be consumed by an arbitrary client as it is served through programming language-independent formats including XML [143] or JSON [182]. Unfortunately, only few online malls like Amazon or eBay offer Web Services for accessing their product catalogs in a structured way. The majority of shops presents offers using traditional HTML pages exclusively. Still, a superior number of those web shops is queriable using information extraction techniques.

To be able to extract product specifications from vendor results pages, a quite complex unsupervised record-level approach could have been developed. However, the retrieval of product information from vendors is not in the main focus. Additionally, since the follow-up algorithms for retrieving product specifications from producer web pages depend on an initial basic product specifications set and, due to given training sets, supervised extraction algorithms are far more reliable than semi-supervised or unsupervised ones, the development of a supervised algorithm was considered to be more appropriate. In the following, wrapper configuration learning as well as wrapper execution for online malls will be presented.

Wrapper Learning

A web scraping wrapper is an adapter transforming information from semi-structured online sources into a structured format, thus enabling machines to process it (Figure 4.5).

![Figure 4.5.: Visual Example for an Information Extraction Wrapper.](image-url)
Wrappers can operate on arbitrary semi-structured sources while their application is especially reasonable for dynamic web pages since such pages often follow a distinct template. Wrappers may be configured for different sources by learning configurations. A configuration for a dynamic website creating results pages that depend on a given query (e.g., search engines or online malls) consists of a query configuration and a results page configuration. Details on both configuration types are provided in the following.

**Query Configuration** Typically, the query configuration is a query URL configuration, that is, it includes general information about the structure of the URL created by the vendor when querying for a product as well as information about each available parameter included in the URL. If a web application is not programmed properly, it might not encode all required information in the query URL. In this case, the wrapper’s query configuration should be a query page configuration and demand a description of the web application’s query page including that page’s address and information about where to find input forms and parameters. Using the query configuration, the wrapper is able to retrieve a results page from a particular website for the given query string.

Query page configurations offer the widest flexibility for interacting with dynamic websites such as online malls and will therefore be used in the following. The pseudo code in Algorithm 4.2 provides a basic idea of the adopted procedure.

**Algorithm 4.2** Detecting Query Page Configurations.

```ruby
1 query_page = request(query_page_url)
2 form_labels = {"query_field" => query_field}
3 paging_labels => {"results_per_page" => 10, "max_results_page_number" => 4,
4   "results_page_number_param_name" => "page"}
5
6 # Find a query page configuration for the given labels.
7 query_page_config = {}
8 form = find_lowest_ancestor_form(query_page, form_labels["query_field"])
9 query_page_config["action"] = form.action
10 query_page_config["method"] = form.method
11 query_page_config["query_param_name"] = form_labels["query_field"].name
12 query_page_config["hidden_params"] = {}
13 form.hidden_inputs.each do |hidden_input|
14   query_page_config["hidden_params"][hidden_input.name] = hidden_input.value
15 end
16 query_page_config["hidden_params"][hidden_input.name] = hidden_input.value
17
18 # Example:
19 # query_page_config = {
20 #   "action" => "search", "method" => "GET", "query_param_name" => "q",
21 #   "results_page_number_param_name" => "page", "results_per_page" => 10,
22 #   "max_results_page_number" => 4, "hidden_params" => {"utf8" => "true"}
23 # }
24 return query_page_config
```
A query page configuration can be generated if the user provides an online mall’s query page and corresponding labels for all relevant fields required to interact with this online mall. Since vendor-specific categories are ignored here, a query text field generally suffices. For more flexible queries, also the results page number parameter name as well as the amount of results per page and the maximum results page number (some online malls only offer a certain number of results pages) need to be supplied by the user. All this information is listed in lines 1 to 4.

In line 8, the enclosing form is detected with the help of the query field. The form’s action and method are saved in the query page configuration in lines 9 and 10. The action represents the endpoint for querying the corresponding vendor while the method is the required HTTP method (in this case, GET or POST). In line 11, the name of the query field is saved since this parameter will be used to forward a user’s request. In lines 12 to 15, the form is examined for additional hidden input fields with fixed values which might be required by a vendor. Each of them is added as a hidden parameter. Finally, the query page configuration is joined with the paging labels from lines 3 and 4. An example of a resulting configuration is given in lines 19 to 23.

With a set of four labels, the described procedure requires a non-neglectable amount of user input to successfully request a vendor’s results page. However, with a growing set of user-given labels, the algorithm learns to guess relevant fields in unknown vendors’ query pages. This is possible due to quite few forms being included in such pages as well as similar parameter naming conventions.

Having the query configuration at hand, the results page configuration can be generated.

**Results Page Configuration** In an HTML context, a results page configuration typically consists of regular expressions, XPath queries, or CSS selectors describing the relevant fields to be extracted. The decision of which technology to adopt relies on the programmer. If a search engine or an online shop tends to change its page structure frequently, regular expressions or CSS selectors might be the better choice as they only rely on small fractions of the path from the document root to the information to be extracted. XPath is to prefer, e.g., when attribute names or CSS information are changed regularly while the structure of the page remains the same. In most cases, the former is true. Compared to CSS selectors, regular expressions certainly allow a much more accurate form of information extraction since they are completely independent of the information’s representation format (in this case, HTML), which allows to distinguish between more and less relevant information contained inside an HTML tag. Anyhow, creating powerful regular expressions can become a rather complex task while CSS selectors are easily created from id or class values contained in the attributes of HTML elements. For example, if a regular expression should cover a specific HTML tag as detailed as possible while the tag’s attributes may appear in varying orders, even capacious tool support could not help to generate fully reliable expressions. Furthermore, today’s online malls generally host professionally designed web pages containing comprehensive style information in the form of CSS. Hence, CSS selectors seem to be the best choice for extracting offers from product search results pages.
The algorithm to learn results page configurations is provided with a results page and the list of tags labeled by the user. These tags are put into groups depending on their type and position in the results page. Each of the resulting groups contains the labeled HTML tags belonging to one result. Such a hierarchical model has also been adopted for STALKER [129]. Then, the algorithm iterates through these groups and finds the common ancestor of each group’s labeled tags. This ancestor is the root element of one contained result. The corresponding CSS selector is saved as the result selector.

Hereafter, each label needs to be observed for creating corresponding CSS selectors. For example, if prices for offer results have been labeled, the algorithm detects a unique CSS selector for the price field inside the HTML content being identified by the previously found result selector. Additionally, unique fields such as the results count, the page number, etc. may be extracted if regarded by the surrounding system.

Again, all information about labeled elements including tag names, corresponding IDs and classes, text contents, etc. is saved to be able to guess potential results page configurations of unknown vendors later on.

It is to be emphasized that the configuration approach presented in this section is not specific to online malls but may be applied to any kind of dynamic website that creates result lists for given queries. Having such a configuration at hand, the wrapper can be executed for actual queries.

**Wrapper Execution**

In the following, the wrapper execution is described briefly. Imagine, a user is interested in the Digi SLR 38 from section 2.2. The query is provided to the system along with some id of which information source is to be used, e.g., Buy.com. The different steps are shown in Algorithm 4.3.

The outcome of line 2 is an offer search results list which has been extracted with a result selector. In lines 7 to 13, the algorithm iterates through the extracted results, fetches all attributes for each offer, and adds the resulting offers to the offers list. The final outcome is a list of maps as described above. This list is independent of the vendor’s layout details and can be provided to a federated search system for further processing.

Unfortunately, a wrapper configuration is only valid as long as the design of an online mall does not change heavily. As mentioned above, the creation of a result page configuration using CSS selectors is more robust in such cases than XPath and easier to generate than regular expressions. Still, it may fail after intensive layout modifications have been effected, causing a federated search system to ask for the creation of a new configuration. Hence, to be more flexible and to offer fallback configurations, the implementation will also contain XPath- and regular expression-based wrappers.

The previous sections presented mechanisms for the federated ranking of products from online malls and extracting basic product specifications out of their results pages. Federated product information systems like the ones presented in section 2.1.2 often do not include the extraction of offers from vendor pages, because a denial-of-service attack reaction might be triggered if many requests are coming from the same IP address. More details on this problem, and how it can be solved, are presented in the following.
Algorithm 4.3 Extracting Offer Search Results.

```ruby
1 results_page = request(vendor_configuration, query)
2 results = select(results_page, vendor_configuration.result.selector)
3 # Select all results contained in the page and extract the offer from each
4 # result.
5 offers = []
6 results.each do |result|
7   offer = {}
8   vendor_configuration.specifications.each do |specification|
9     offer[specification.name] = select(result, specification.selector)
10    end
11   offers << offer
12 end
13
14 # Example:
15 # offers = [
16 #   { "name" => "Digi SLR 38", "producer_name" => "Digi", "price" => "$499" },
17 #   { "name" => "Digi SLR 38 Camera Case", "producer_name" => "Digi", "price" => "$29.99" }
18 # ]
19 return offers
```

Denial-of-Service Attacks

When being faced with the fact that online information is not provided in a structured way, the mechanisms described above offer the only possibility to make it machine-processable. Unfortunately, accessing web pages programmatically through a single query interface that can be used by different users and applications may introduce another issue, namely, the reaction to potential Denial-of-Service (DoS) attacks. Such attacks flood a web application's network with an unexpected amount of packets or overly large packets and may be defeated by limiting the amount of requests per IP address and time unit. Distributed DoS (DDoS) attacks have been used successfully on the websites of PayPal, Visa, and MasterCard [166]. General information about both types of attacks including design decisions in the Internet that created the potential for (D)DoS attacks as well as how they can be defeated is to be found in Peng et al. [140].

A service offering a unique vendor query interface to arbitrary consumers accepts requests for product offer lists and retrieves offer results pages corresponding to the given queries from the vendors of interest. Thus, since especially online malls have highly dynamic websites, and caching strategies can only be applied in a quite limited manner, the majority of requests to this service is followed by requests to configured vendors. From the vendors’ point of view, all requests are arriving from the same IP address which
might provoke reactions on potential Denial-of-Service attacks on the vendors’ side.

The idea for avoiding reactions on potential DoS attacks consists of letting the web service consumer execute the vendor page requests on her own. This can be achieved programmatically by offering the query page configurations for the different vendors in a machine-readable format to the consuming entity. The consumer may either request the pages using these configurations or, if it is a web application itself, delegate this information one step further to the client which might be a browser or a mobile client. The corresponding communication paradigm and an actual implementation of a plugin for standard browsers as well as a mobile iOS client will be described in section 6.1.5 and 6.1.6 of the evaluation chapter.

Having all previously described techniques at hand, a complete system for federated product offer search can be designed. Apart from providing effective product offer search to potential consumers, such a system delivers bootstrapping information to additional algorithms that aim at creating all-embracing product information sets consisting of technical specifications. The concepts for retrieving and extracting such specifications are presented in the next sections.

4.2. Producer Product Information Search

From the consumer’s point of view, product specifications provided by producer websites are the most important product information as they create a detailed view on the product of interest and make it comparable to related products. Concerning the FEAD Chain introduced in chapter 3, locating and extracting such information to enhance an initially created basic product information set belongs to the enriching step (Figure 4.6).

![Figure 4.6.: The FEAD Chain - Enrich Basic Product Information.](image)

As depicted in the introductory chapter (chapter 1), federated consumer product information systems still depend heavily on the manual submission of product information by employees of the corresponding information system. Thus, this section focuses on
automating the process of locating and extracting product information provided by producers. Preliminary results of this section have been published in [176] (see also [83]).

4.2.1. Producer Product Document Retrieval

For the retrieval of product specification pages from producer sites, a set of requirements has been stated in section 1.4.3. To simplify matters, they are restated here.

**Req 1.1** The locating of producer pages should only be based on a product’s name as well as its producer’s name.

**Req 1.2** The locating algorithm should also work without hints on where to find the page.

**Req 1.3** Even if different producer product web pages are available, the algorithm’s output should be the actual product specification page.

As defined in Req 1.1, the document retrieval component has to find the product specification page provided by the producer with the minimal input of a product name and its producer’s name. The success of this action is of fundamental importance to the follow-up algorithms as no information can be extracted without having such a document at disposal.

The document set to consider is the total number of publicly available web pages $W$. Let the product whose specification page is to be found be $x$. Thus, all web pages presenting information about this product can be subsumed as $W(x)$. Since only specification pages are of interest, these web pages are defined by $W_S(x)$. Specification pages may be distributed all over the Web and offered by arbitrary sources. However, product manufacturers are accounted to be the most trustable sources concerning their own products. All web pages provided by a manufacturer producing $x$ can be summarized by $W(m(x))$. Hence, the document to be found is one of the web pages $W(m(x)) \cap W_S(x)$. In the majority of cases, only one producer’s specification page exists per product (and presentation language), therefore following through with $|W(m(x)) \cap W_S(x)| = 1$. If so, this page is curtly defined as $w_x$.

The formula shows that the DR component’s task consists of determining the set of producer web pages $W(m(x))$ for the producer of $x$, filtering out the set of pages presenting information about $x$, and finally detecting $w$ or choosing one of the found product specification pages. Thus, the retrieval is laid out as a two-step process. In a first step, the producer page is located and, in a second step, the product specifications page is searched restricting the requests to the producer domain.

One or more search engines are required to execute the described queries. Thus, a possibility would be to set up a dedicated search engine that is fed by a focused crawler, specifically adapted to the task of product page collection from producer pages. The search engine and its associated crawler were to be built following the principles laid out in section 2.1.1. However, the crawler would require a list of products and corresponding producers for being able to find specification pages. Such a list would certainly not be complete and additionally require steady updates. Furthermore, the crawler would need
a vast amount of network and processing power to only cover a part of the enormous product domain. In the end, it would still not be able to compete with big players such as Google, Bing, and Yahoo offering their index contents as a free service to everyone. Thus, their web search services are to be employed in the retrieval component instead.

An overview of the retrieval process is given in Figure 4.7. In steps one to three, a list of potential producer pages is retrieved taking the producer’s name into account. Then, based on heuristics to be explained below, the correct producer is chosen in step four. With the producer’s domain and the product’s name at hand, potential product specification pages are retrieved in steps five to seven and, finally, the correct specification page is chosen in step eight. More details are given below.

Figure 4.7.: Product Specifications Page Retrieval Overview.

**Producer Page Retrieval**

The input of the producer page retrieval procedure is the name of \( m(x) \), e.g., “Nikon”. The result is the top-level domain of the host and one further level. For example, if the URL “http://www.nikon.com/” is detected to be the main producer web page, the domain name “nikon.com” is returned.

Hence, in a first step, different public search services are queried with \( m(x) \). The results returned by all search engines are ordered using Borda Ranking [118]. Borda Ranking is an algorithm for generating integrated result lists composed of several ranked result lists. A ranked list consists of \( n \) ordered elements. The best-ranked element receives \( n \)
points, the second element \( n - 1 \) points and so forth. The scores from all ranked lists are summarized for each element and thereby the integrated ranked result list can be created. Finally, the element with the highest score is chosen to be the producer website candidate. For example, if the third result of 87 results retrieved by Google is inspected, a score of 85 is added to the overall Borda Ranking value of this result.

To enable searching on the producer’s site, the producer domain is extracted based on the Public Suffix List [65]. If the domain has not been visited before and it is not blacklisted (e.g., “wikipedia.org”), it is provided to the product specification page retrieval component. As long as that component cannot retrieve the product page in the given domain, the domain of a result with a lower Borda Ranking score is returned.

**Product Specifications Page Retrieval**

For locating the actual product page \( w_x \), again different web search services are queried (steps one to four in Figure 4.8), this time using the product’s name as query and restricting the search space to the retrieved producer domain. Thereby, the potential result set reduces from \( W \) to \( W(m(x)) \). Since this set may not directly show the product’s specification page as the first hit, a series of ranking algorithms is to be applied. The whole process is visualized in the bottom part of Figure 4.8.

![Figure 4.8.: Scoring Potential Product Specifications Pages.](image-url)

Emerging search result lists are combined using the Borda Ranking algorithm described in the producer page retrieval section (steps six and seven). Unfortunately, the retrieved
result sets of the different search engines are filtered by the search engine providers, e.g., by removing near-duplicate addresses. This way, different product pages like general description pages, specification pages, etc. are merged into one search result. Hence, it is possible that the actual product specification page is not in the result set and has to be retrieved explicitly. In steps eight and nine each result page taken into account (e.g., the first ten results) is therefore requested and scanned for product specification links. All links’ texts are matched with a set of characteristic link text patterns. Finally, for each search result, the best-matching link is added to the original result set inheriting its referencing page’s Borda score if it had not yet been included. Additionally, a specification score is assigned to this URL that is calculated based on the matching success. Since this process is the most interesting one, Algorithm 4.4 provides some pseudo code.

Algorithm 4.4 Calculating Specification Ranks.

```
1 search_results = borda_rank(search_results_lists)
2 link_text_pats = [/Product Specifications/i, /Specs/i, ...]
3
4 # Iterate through all links contained in a search result’s web page and
5 # detect the similarity of the links regarding given link text patterns.
6 search_results.each do |search_result, rank|
7   candidates = []
8   links = extract_links(get_page(search_result))
9   links.each do |text, url|
10      link_text_pats.each_with_index do |pattern, index|
11         if(pattern =~ text)
12            text_similarity = pattern.match(text).size/text.size
13            pattern_specificity = (link_text_pats.size-index)/link_text_pats.size
14            candidates << [text, url, text_similarity*pattern_specificity]
15         end
16      end
17     end
18     search_results << best_link(candidates)
19   end
20
21 return search_results
```

As the code shows, for each link in a search result’s page the patterns are compared with the link text (line 11). Depending on the relative length of the detected pattern (line 12) as well as its specificity (line 13; the given link text patterns are ordered by expressiveness), a specification rank is calculated for the best link (line 14). During runtime, the component may learn additional significant link text patterns.

After completion of the described result list extension, each result is evaluated concerning its URL (URI path ranking in steps 10 and 11 of Figure 4.8) and its referenced page’s title (title ranking in steps 12 and 13) and content (content ranking in steps 14 and 15). The URI path score is based on the appearance of positive or negative keywords being characteristic for product specification pages in the result’s URL. Thus, terms like
“product” or “specification” would augment the URI path score while “forum”, “news”, or “press” were adverse for gaining a high score. Additionally, substrings of the product’s name are searched in the path increasing the score if being found. The title score is based on the appearance of product name substrings in the title of the web page belonging to the examined result. That is a feasible approach as untruly found specification pages belonging to different products receive a lower rank this way. The assigned score then depends on the percentage of matching product title substrings. In a last step, a content score is calculated. This depends on the appearance of known attribute key phrases from former extraction procedures. The system retrieves all these phrases from a database, matches them with the text nodes of each result’s web page, and calculates the percentage of keys found in the page. Based on this percentage, the content score is assigned. Finally, all scores are combined with the present result scores to receive a final ranked result list. The first result is chosen to be the product specification page and is provided to the information extraction procedure. An example of how a ranked product specifications page list might look like is given in Table 4.2.

Table 4.2.: Example Scores Calculated by the Ranking Algorithms.

<table>
<thead>
<tr>
<th>Document</th>
<th>Borda Rank</th>
<th>Spec. Rank</th>
<th>URI Rank</th>
<th>Title Rank</th>
<th>Content Rank</th>
<th>Overall Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>/products/cams/slr38</td>
<td>18</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>/products/support/slr38</td>
<td>15</td>
<td>0</td>
<td>-2</td>
<td>9</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>/products/cams/slr38/description/specs</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>46</td>
</tr>
<tr>
<td>/products/cams/slr38/description/features</td>
<td>11</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>/support/software/slr38</td>
<td>9</td>
<td>0</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

If the described algorithm has successfully located a web page that is offered by the current product’s producer and presents technical specifications for the product of interest, the next follow-up step, namely, the extraction of product specifications out of this page, can be executed.

4.2.2. Producer Product Information Extraction

A set of three requirements has to be fulfilled by the extraction algorithm (section 1.4.3). For simplicity reasons, they are provided here again.

**Req 2.1** The extraction routine has to be able to extract information when being supplied with only one product specifications page.

**Req 2.2** When no knowledge from previous extractions is given, the algorithm still has to be able to identify the extraction targets.
Req 2.3 Independent of the actual page template, the results of the extraction process should be a list of product specifications adhering to the producer’s terminology.

As depicted in chapter 2, current methods for information extraction like DEPTA [189] and ViPER [160] already allow the information extraction from few similar pages. Req 2.1 cannot be fulfilled by those approaches yet. Different page templates (Req 2.3) are also not considered by existing work. Thus, the contributions of this chapter are techniques to fit all requirements mentioned above.

The goal of the information extraction process is to gather technical product specifications from a given specification page. Product specifications may have sundry structures. Typically, they consist of a series of key-value pairs, each describing one product feature. For example, the resolution of a digital camera lens might be given by the key-value pair “Effective Pixels: 8.2MP”. “Effective Pixels” represents the key of the given feature whilst “8.2MP” is the feature value. The separation of key and value is achieved by putting a separator between them (in this case, a colon). As all presented algorithms operate on the public Web, the specifications are provided through HTML pages. Hence, key and value of a given feature might also be separated by HTML tags. Both alternatives are processable by the algorithms presented below. Unfortunately, in rare cases, features do not include a key phrase which prevents an expedient mapping. An examination of the utilization of such features will only be taken into account partly. Specifications provided in binary formats, such as graphics or Flash files, will not be examined at all.

The analysis of related work in section 2.2.3 pointed out that current approaches like NET [120] and ViPER [160] do not only rely on a web page’s source code for generating good results, but apply visual information during the extraction process as well. Hence, the algorithms to be developed in the following use a combination of structural web page properties like element XPath queries as well as element coordinates and visibility settings to detect the actual product specifications in a page.

As mentioned before, the information extraction component is only provided with a product specifications page and tries to extract the specifications from this page. Concerning the previously examined feature structures, key and value are expected to share similar XPath queries. The goal is to find those and provide them to a web scraping wrapper. The wrapper can then be used to deliver the specifications to the product information system. An overview of the procedure is given in Figure 4.9.

In steps 1 to 4, the specification page is retrieved through a Web Page Analyzer and XPath queries for all page elements are created. Additionally, the analyzer renders the retrieved page and extracts information about the included elements’ spatial arrangement. This includes the coordinates, the contained text, and the visibility of each element. Having the set of web page elements with all this information at hand, the extractor provides them to a wrapper configurator in step 5. The creation of a wrapper configuration consists of four main tasks. In the first task (steps 6 and 7), all elements are clustered into lists based on given criteria. Some example criteria are given in the figure. An element list is supposed to include elements of similar type, e.g., keys of the desired product specifications. The element lists are then purged and insignificant clusters are dropped. During the second task (steps 8 to 9), element lists are clustered into groups.
A group includes element lists being similar to each other. In the best case, one of the groups contains the elements list with the product specification keys and the elements list with the values. Again, created groups are purged and insignificant ones are dropped.

The third task (steps 10 to 11) consists of creating candidates, that is, the cluster of groups potentially including the group with the product specifications. Based on the best-rated candidate, the wrapper configurator can finally create a set of XPath queries for extracting product specifications and provide these queries to the wrapper component. The wrapper component extracts the actual product specifications and delivers them to the extractor (step 13). An example for the clustering functionality is visualized in Figure 4.10.

Figure 4.10.: Clustering of Product Web Page Content.
As can be seen, the algorithm first chooses four element lists in this case, namely, $L_1$, $L_2$, $L_3$, and $L_4$. The purging and dropping steps might already filter out $L_1$ and $L_2$. If not, the group creation step might create $G_1$ and $G_2$ as potential product specification lists. If the clustering algorithm is adequately configured, $G_2$ is chosen to be part of the final candidate. All important stages of the extraction process (web page analysis, clustering, wrapper configuration, wrapper execution) will be examined in detail in the following. Additionally, a learning component will be described.

**Web Page Analysis**

Having the source code of a web page at hand, a set of different programming libraries allow the analysis of this code based on the HTML tree. For example, XPath queries or CSS selectors for different elements can be generated. Web page elements may be clustered based on similar XPath queries and their contained texts. However, the resulting element clusters may, e.g., contain lists of strings not visible to a user when the web page is actually rendered in a browser or lists with horizontal alignment while product specifications are typically represented by vertically aligned lists. It is thus indispensable to take the actually rendered web page into account. This can be accomplished by using a GUI-less browser or libraries for the programmatic access of standard browsers. ViPER based its visual analysis on JRex \[67\] (Java Browser Component) which uses an old Mozilla browser for web page analysis. Meanwhile, quite feature-rich alternative libraries emerged, allowing more complex interactions with web pages (see section 6.1.7).

Executing a set of JavaScript functionalities against a rendered web page through such a library retrieves the coordinates $x_{\text{min}}$ (left), $y_{\text{min}}$ (top), $x_{\text{max}}$ (right), and $y_{\text{max}}$ (bottom) of each element. The contained text and the visibility of the elements can be gathered as well. An element’s visibility depends on its visible attribute setting as well as its coordinates. If all coordinates equal zero, the HTML element is not visible. An example structure for an HTML element in XML representation might look like in Figure 4.11.

```
<element>
  <left type="integer">8</left>
  <top type="integer">85</top>
  <right type="integer">60</right>
  <bottom type="integer">104</bottom>
  <visible type="boolean">true</visible>
  <text>Canon</text>
</element>
```

Figure 4.11.: Example Representation of an HTML Element in XML.

For each element, its rooted XPath query, its coordinates, its visibility, and its text are included. The web page analyzer finally returns all elements which are contained in the provided HTML page. The clustering step employs this information in the next step.
Web Page Clustering

As described above, three clustering tasks are to be completed for finding product specifications in a given web page, namely, the creation of element lists, the merging of such lists to obtain groups, and the generation of candidates. However, these clustering tasks only differ in terms of cluster granularity. Conceptually, the idea is always the same. Hence, before going into peculiarities of the single tasks, the generic clustering process is to be described using the abstract term *item* as an instance of clusterable types and *cluster* as the conjunction of such items.

To be able to generate meaningful clusters, a cluster measure is required that decides which items should be combined to build a cluster and which items do not belong to that cluster. This can be established by using dedicated cluster IDs. A cluster ID is determined by a set of properties having identical or similar values for all items contained in the cluster. For every new item, a cluster is extended by this item if an ID being created based on the item’s properties matches the cluster ID. If the item ID does not match any cluster ID, a new cluster is to be created with that item’s ID and the item is inserted into the new cluster. The general procedure is shown in Algorithm 4.5.

**Algorithm 4.5** Cluster Creation Based on IDs.

```ruby
1 clusters = []
2 opts = {:type => "cluster", :include_text => true, :visible => true}
3
4 items.each do |item|
5   # Create an ID for each item and find clusters matching this ID.
6   item_id = create_id(item, opts)
7   cluster_indices = []
8   clusters.each_with_index do |cluster, index|
9     if(item_id.match(cluster.id) || cluster.id.match(item_id))
10        cluster_indices << index
11     end
12   end
13
14   # Either add the item to available clusters or create a new cluster.
15   if(!cluster_indices.empty?)
16      cluster_indices.each do |cluster_index|
17        clusters[cluster_index] << item
18        clusters[cluster_index].id = create_id(clusters[cluster_index], opts)
19      end
20   else
21      cluster = Cluster.new(item_id)
22      cluster << item
23      clusters << cluster
24   end
25 end
26
27 return clusters
```

---

*Producer Product Information Search*

103
In line 2 some options are provided that decide about how the cluster IDs should look like (type, include_text, and visible). type is only used to differentiate between lists, groups, and candidates. The other two options cause four clusters to be created, namely, a cluster of invisible items without text, a cluster of invisible items including text, a cluster of visible items without text, and a cluster of visible items including text. In line 6, an ID is created for the current item. Thus, all clusters matching this ID can be collected in lines 8 to 12. If at least one matching cluster has been detected, the item is inserted into the found clusters and their IDs are updated in lines 16 to 19 (The update is necessary as a new item might influence the cluster’s ID). Otherwise, a new cluster is created for that item in lines 21 to 23. An example ID for the options provided in the pseudo code is given in the following:

```
?-mix:type="cluster" include_text="true" visible="true"
```

The cluster described by this ID includes visible items that contain text. include_text and visible are only two possible criteria for ID creation. For a complete list of element and cluster criteria, please consider the Appendix.

The prefix “?-mix:” states that the IDs are created as regular expressions. m stands for “make dot match new lines”, i for “case insensitive”, and x for “ignore whitespaces”. In the given ID, all three options are deactivated. Regular expressions make the clustering process more powerful as the next sections will prove. For being able to use characters having a certain meaning in the context of regular expressions, every such character has to be potentially escaped if not belonging to a regular expression part of the ID.

The clusters created when running the described algorithm contain items with similar or identical values for the chosen criteria. They might, however, still not be as clean as desired by the extraction component. Thus, a purging step allows the clusters to rearrange their items. This might also include splitting up themselves into smaller clusters. The clusters could, e.g., be instructed to split themselves into several new clusters if their text contents belong to different languages.

Finally, clusters may also be dropped based on given options. For example, if a cluster’s items all contain the same text, the cluster could be removed from the overall cluster set.

The idea of clustering web page elements has already been adopted in previous works. Álvarez et al. [5] used XPath queries to put web page elements into clusters. Some other works utilize HTML element attributes to create meaningful clusters. However, the combination of several different properties from a document’s tokens, its tree representation, and visual information for creating clusters is an entirely new approach. The described mechanism allows a flexible, parameter-based creation of clusters that may be adapted for various extraction tasks. The purging and dropping steps take care of cleaning up the resulting cluster set. In the following, the algorithm will be configured to adequately extract product specifications by the use of an extraction plan consisting of the initially mentioned tasks. Therefore, peculiarities of each step are given in the next sections.

**Clustering Elements** Clustering elements based on a set of properties is the first task to be executed on the contents of a web page. A typical property set for creating element list
IDs consists of type, include_text, indexless_xpath, maximum_left, and visible. Hence, a resulting example list might include all elements that contain text, share the indexless XPath query "/html/body/div/a", have the maximum left coordinate “4”, and are visible. The maximum left coordinate proved to be a valuable feature. It describes the \( x_{\text{min}} \) coordinate of the highest ancestor of an HTML element that does not contain additional text in comparison to the one already included in the current element. For example, if the product specification keys are wrapped by <b> tags with preceding empty <div> tags of variable length, the maximum left value is still identical for all key elements and hence allows to put those keys into the same cluster. An example ID for a corresponding cluster is given in the following.

\[
(\text{-mix:} \text{type="list" indexless_xpath="/html/body/div/a" include_text="true" maximum_left="4" visible="true"})
\]

As mentioned above, the IDs are represented as regular expressions. Therefore, all slashes in the indexless XPath query are to be escaped.

A typical property applied during the purging step is split_by_std_dev_text_length. This criterion needs to be supplied with a value that determines the maximum ratio the standard deviation of the list’s element text lengths may have in comparison to the maximum difference of text lengths.

Available options for dropping element clusters include is_not_noise, has_min_size, has_alphanumeric_text, or has_varying_text. is_not_noise is provided with a list of HTML tag names that should not be included in the elements’ XPath query (e.g., <option>). The other criteria are comprehensible through their names.

The resulting element lists represent clusters from all over the given web page. If the criteria for clustering, purging, and dropping have been chosen elaborately, a list of product specification keys and a list of product specification values are part of the list set. Depending on the representation format, keys and values might also be contained in one list. It is the goal of the list clustering task to assign the keys list and the values list to each other if they are contained in different clusters.

**Clustering Lists** When clustering lists of web page elements, the result is a number of groups including element lists that belong to each other according to the chosen criteria. Lists of product specification keys and values are generally located at similar positions in the web page of interest. More specifically, they often share similar \( y_{\text{min}} \) values or have close to identical heights. A typical property set for creating group IDs thus includes type, regex_average_top_range, or regex_shortened_indexless_xpath. The average top range of a group consists of a set of \( y_{\text{min}} \) coordinates around the average \( y_{\text{min}} \) coordinate for that group. The average is calculated by taking all \( y_{\text{min}} \) values of included items, in this case, lists, into account. The prefix “regex” describes the ID part as to be a regular expression. Thus, it is not escaped. Also, the shortened and indexless XPath query is marked as to be a regular expression. Both criteria are extended with “*” by their respective functions for allowing other IDs to have additional elements at these positions and still match with the ID. An example of a resulting ID is shown in the following.
The power of adopting regular expressions can be seen when the ID is compared to an ID including \texttt{regex\_shortened\_indexless\_xpath="/html/body/div\/*"} and \texttt{regex\_avg\_top\_range=".*(22|23|24).*"} which would still match and thus allow putting both items into one cluster with a subsequent cluster ID update.

When purging the generated groups, \texttt{keep\_biggest\_items} and \texttt{sort} are helpful criteria. The first criterion removes the smallest element lists from each group until a certain quantity of included lists is reached while the second criterion puts included lists into a given order. The order might be \{\texttt{:top => "asc"}, \texttt{:left => "asc"}\}, thus sorting all lists by their $y_{min}$ and $x_{min}$ coordinate in ascending order.

Finally, groups are dropped that do not adhere to a set of given criteria. For example, groups with less than two element lists might be removed (\texttt{has\_min\_size}). The next step joins groups to create candidates.

**Clustering Groups** Element list groups already include potential pairs of product specification keys and values. The final clustering step thus only identifies candidates out of the available groups. This is accomplished by putting all available groups into one cluster. The candidate may be purged by \texttt{keep\_biggest\_items} if groups with few element lists are not of interest.

A more effective purging criterion can be applied if a user provides one or more product specifications contained in the examined web page. Such a specification may consist of a key “Optical Zoom” for the digital camera domain and a corresponding value. In this case, the clustering method, and thus the extraction procedure, is to be categorized as a supervised one since the user implicitly labels the given product page. When such examples are provided, the clustering algorithm looks for the example phrases in the available groups and chooses the correct one to contain the specification keys.

Assuming that the whole information extraction process was implemented in a product information crawler, the crawler could not wait for a potential user to provide product specifications from each found page. The extraction system thus has to fall back to a more generic method. The system may already have gathered product specifications during previous routines which are similar to the ones contained in the current web page. These specifications are stored in a database and may be of use for the current page. Depending on the prior knowledge being confirmed by a user or not, such an approach is to be categorized as semi-supervised or unsupervised, respectively. As shown in section 2.2.3, such algorithms are of high interest in current IE research works. Available product knowledge in the form of extracted product specifications may be taken into account during the purging step using \texttt{sort} in combination with the amount of found key phrases per group. With a growing information base, it is very helpful to use this knowledge for enabling better extraction results.

The complete process may also run without domain knowledge by providing an adequate initial configuration. In this case, it is certainly an unsupervised approach.
As the clustering sections showed, the applied algorithm is extensively configurable. Choosing the right configurations for each clustering step is of utmost importance since the quality of the extraction results is fully dependent on the correct groups candidate, even if user-supplied or prior knowledge is available during the clustering process. When no training set of product web pages and corresponding extraction results is available, a manual configuration is possible. However, gathering training sets allows automating the configuration creation through a learning approach. The next section will describe such an approach.

**Learning Clustering Configurations**

With a simple trial-and-error method it is possible to detect the most important clustering criteria manually. Some of these have been mentioned in previous sections. Anyhow, details like the exact value of an applied threshold (e.g., `split_by_std_dev_text_length`) or the range of a coordinate (e.g., `regex_avg_top_range`) are only hard to predict manually. Especially side effects on one criterion if another one is changed are difficult to foresee and may influence precision and recall of the final extraction results. Therefore, a method for easing the configuration process is presented in the following. The code in Algorithm 4.6 describes the general configuration learning process.

**Algorithm 4.6 Learning the Optimal Clustering Configuration.**

```
1 CONFIGURATIONS = {
2   :create_lists => {:type => ["list"], :include_text => [true], ...},
3   :purge_lists => {:split_by_std_dev_text_length => [nil, 0.8, 0.85], ...},
4   :drop_lists => {:is_not_noise => [nil, ["select", "option", "a"]], ...},
5   :create_groups => {:type => ["group"],
6                      :regex_avg_top_range => [nil, 200], ...},
7   :purge_groups => {:keep_biggest_items => [nil, 2],
8                      :sort => [{:right => "asc", :top => "asc"}, {:bottom => "asc"}], ...},
9   :drop_groups => {:has_min_size => [2], ...},
10  :create_candidates => {:type => ["candidate"], ...},
11  :purge_candidates => {:keep_biggest_items => [nil, 1],
12    :sort => [nil, {contained_key_phrases => "desc"}], ...},
13 }
14
15 # Create all possible configurations out of CONFIGURATIONS and save the
16 # configuration returning the best F1 score.
17 configuration = initial_configuration(CONFIGURATIONS)
18 f1_score = 0
19 begin
20   new_f1_score = cluster_and_extract(training_set, configuration)
21   save(configuration) if(new_f1_score > f1_score)
22 end while(configuration = next_configuration(configuration, CONFIGURATIONS))
```

The major work of the algorithm consists of finding the different configurations out of the general CONFIGURATIONS variable (next_configuration()) as well as clustering
Product Information Extraction from the Web

web page elements, generating wrappers, and extracting product specifications (cluster_and_extract()). Since including corresponding pseudo code for these steps would be quite confusing, the focus has been laid on the main idea of the learning algorithm. Lines 1 to 13 offer an example for a coarse configuration being provided by the user. As can be seen, some criteria are already definite (type, include_text, has_min_size) since the lists of values they are pointing to only include one element. The other criteria offer various values (e.g., the first sort either wants the groups' items to be ordered by right \((x_{\text{max}})\), then top \((y_{\text{min}})\) or by bottom \((y_{\text{max}})\)). If a criterion's values include false or nil, the criterion may also not be applied in the possible configurations.

In line 17, the initial configuration is created out of CONFIGURATIONS. This configuration includes all criteria, each pointing to the first of its possible values. In lines 19 to 22, the clustering and extraction steps are executed for the given evaluation set of HTML pages and extracted specifications. If the current configuration retrieves product specifications at a higher \(F_1\) score than the last one, the configuration is saved. The process continues until all possible configurations have been tried out.

As can be seen, the configuration learning approach is a kind of multi-criteria analysis. The complexity of the algorithm is exponential concerning its input parameters since increasing a value list by one doubles the amount of possible configurations. This fact can be mitigated by classifying the configuration parameters as impairable and non-impairable ones. Non-impairable parameters may be optimized independently, thus making the learning approach complexity linear. Then, with the non-impairable parameters set, the impairable configuration parameters can be learned using diligently chosen candidate values. A configuration like the one given in the example above produces 192 possible configurations. is_not_noise, keep_biggest_items, and sort can be classified as non-impairable. Therefore, the amount of configurations to be checked reduces to six.

With a powerful clustering configuration at hand, the correct product specifications candidate may be picked out of all created groups of a given product page. In the next step, a wrapper configuration consisting of a set of XPath queries is to be generated.

Wrapper Configuration

Being provided with an extraction candidate, the task of the wrapper configuration component is to derive a set of XPath queries out of the group's element lists. As mentioned before, two major occurrence forms have been found for the representation of key-value pairs in web pages. In the first form, a simple character string is used to separate key and value (e.g., a colon). In the second one, key and value are separated by HTML tags. Additionally, when having HTML-separated key-value pairs, key and value may share a common ancestor that does not have any other keys or values as successors. In few cases, all keys share a common ancestor and all values share an ancestor.

The algorithm first assumes the keys and values to be separated by a simple character string. Therefore, the best-ranked elements list is examined and its text contents are scanned for previously defined separators. If the number of element texts containing such a separator exceeds a certain threshold, the algorithm calculates the common XPath query. A common XPath query has at least equal tags (but not necessarily equal indices)
at each position for all given XPath queries. Varying indices at previous positions are
simply stripped. The common XPath query can be returned together with the detected
separator. An example is given in Figure 4.12.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Key XPath Q.: /td[1]/b[1]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.12.: Important XPath Queries for the Wrapper Configuration.

If no separator could be detected, the algorithm assumes keys and values to be separated
by HTML tags having common ancestors for each pair. Thus, a specification XPath
query is calculated by splitting the common XPath query on the last eliminated index.
The relative key XPath query is built by choosing the remaining XPath query that should
be identical for all elements. The combination of both selects all product specification
keys. The specification values are found by joining all texts of elements identified by the
specification XPath query and removing the keys from these.

If no values can be detected using this method, the configurator assumes key-value
pairs to not have a common ancestor being different from the other pairs’ ancestors.
Therefore, the second elements list in the chosen group is expected to contain the values.
A new specification XPath query is calculated that is valid for all elements of the first
and the second list. Then, the key XPath query is built by choosing the remaining XPath
query of the first elements list. The value XPath query is determined by saving the
remaining XPath query of the second elements list.

With a wrapper configuration, that is, a set of XPath queries, at hand, the actual
extraction step can take place which is to be described in the next section.

Wrapper Execution

Executing the process of web page analysis and elements clustering already returns a set
of product specifications and does not take too much time. It might thus also be executed
repeatedly for the same web page without saving the resulting wrapper configuration.
Configuring and executing a wrapper still makes sense for several reasons.

First of all, product specifications may be scattered all over the web page of interest.
A clustering mechanism being supported by visual properties tends to only extract one
specifications list from the product page. Having an XPath-based wrapper configuration
at hand, all product specifications being visually located in different lists may be extracted
in most of the cases. Additionally, producers often reuse templates for different products.
It is thus worthwhile to first try out existing wrappers (that might even be confirmed by
potential users) from the current producer in order to improve the extraction quality.
Lastly, executing a set of XPath queries is still faster than analyzing a web page and clustering its contents, especially when adopting regular expressions. This will be shown in the evaluation chapter.

The extraction procedure itself has virtually been described in the previous section. The common XPath query is used to find the set of product specifications. For each element found, the contained texts are split by a detected separator. If no such separator has been found, key XPath query and value XPath query are added to the common XPath query to find keys and values, respectively. If no value XPath query is available, the contents found with the common XPath query minus the key’s text are accepted as product specification values.

Having the described suite of algorithms for retrieving product specification pages as well as extracting contained specifications in a supervised, semi-supervised, or unsupervised manner at hand, the process of gathering such high-quality specifications can be fully automatized. However, although product specifications offered by producers are highly reliable, they differ in many dimensions including terminology, complexity, and adopted units. Thus, a follow-up process must integrate the given specifications with a central knowledge model to make the product information comparable. Before the next chapter will present concepts of product specifications integration, a short section will consider ideas for third-party product information search since it might offer potential consumers valuable additional information.

### 4.3. Third-Party Product Information Search

Third-party product information sources include all information sources which are not product vendors or producers. Test pages, boards, blogs, search engines, etc. belong to this group. Information from such sources is often important for users since it is created by other consumers.

Third parties can be divided by their dynamics into dynamic and high-grad dynamic sources. Dynamic third-party sources include the mentioned test pages, boards, and blogs. High-grad dynamic third parties include web applications like search engines of different natures. Accessing dynamic third-party sources through a programmatic interface requires intensive information processing, e.g., analyzing sentiments in contained product user opinions. Sentiment analysis is a research topic on its own and will only be mentioned briefly in the outlook (section 7.3). High-grad dynamic third-party sources such as Google or TextRunner [187] may be handled similarly to the online malls presented in the beginning of this chapter. The algorithms for wrapper learning and wrapper execution can be used for third parties one-to-one since they are adoptable for all kinds of high-grad dynamic web sources offering search and results pages. The ranking of third parties cannot be transferred that easily since such sources generally lack a set of commonalities that could be exploited as ranking characteristics.

However, as already mentioned in the introduction chapter, third parties are not the focus here. The implementation of third-party information search will be described sparsely in the evaluation chapter while no continuative concepts are to be developed.
4.4. Conclusions

The current chapter dealt with the federated search of product information in different source types. For each source, present peculiarities have been inspected and used for developing adequate algorithms to retrieve web documents and extract valuable information from them.

The first section presented a series of algorithms for the domain of vendor sources. Extracting offers consists of learning wrapper configurations for different vendors. Such configurations include query configurations for retrieving results pages and results page configurations offering the possibility to extract single search results being composed of a set of attributes, i.e., product specifications, out of those pages. The idea of how to avoid potential reactions to assumed denial-of-service attacks has been presented as well. Additionally, a ranking algorithm has been developed in this section. It is composed of three different steps, namely, vendor ranking, query categorization, and result ranking.

The second section offered algorithms for retrieving and extracting product specifications directly from the products’ producer pages. Such information is estimated to be of eminent usefulness since it describes (mostly technical) details of examined products. The first step here, namely, the product specifications page retrieval, consisted of identifying the producer’s domain through the use of different search engines and detecting the product specifications page on that domain. Hereafter, the specifications extraction can take place. Again, first a wrapper has to be learned which can happen in a supervised, semi-supervised, or unsupervised manner. Having an adequate wrapper at hand, product specifications can be extracted.

The developed algorithms enable the creation of an all-embracing view on arbitrary technical products by gathering information from a series of relevant sources. However, although the algorithms have been specifically adapted to the product domain, their general functionality can also be applied to other domains. For example, one could imagine a service for the federated collection of job offers which is what Rapier [27] was developed for. Web pages presenting such offers through the use of a search functionality could be configured in this service to always deliver the job details in the same format while the job portal executes a ranking mechanism being similar to the one presented here. Another example is a service that finds information about people from social networks. The capacious configurability of the cluster-based extraction algorithm would allow the comfortable extraction of details about persons from a set of different networks without adapting the algorithm to each site. A configuration could even be optimized automatically by applying the learning algorithm presented above.

The main research focus lies on product specifications gathered from producers’ product pages. Therefore, the next chapter will develop algorithms for the integration of such specifications with a central knowledge model that may be represented by an ontology.
The previous chapter presented a complete approach for retrieving, extracting, and ranking product information from different product information sources. As already pointed out in the introductory chapter, the most important information is to be retrieved from producer sites since this information makes products comparable. Section 4.2 points out how to locate and extract such information with only very little user interaction. After execution of the described steps, a significant information set can be presented to the user that offers an essential simplification of the entire purchasing process. Unfortunately, the presented specifications are retained in a terminology that is specific to the particular producer manufacturing the product of interest. This fact makes it difficult for consumers and impossible for machines to compare different products effectively. Thus, the present chapter aims at developing concepts for enabling the product comparison by machines having the heterogeneous product information from different producers as a starting point. All included techniques belong to the administration step located at the third position of chapter 3’s FEAD Chain (Figure 5.1). Hence, product ontologies, product ontology matching, and machine learning approaches will be relevant throughout this chapter.

Figure 5.1.: The FEAD Chain - Manage Product Information.
The developed approach has to fulfill a set of three coarse-grained requirements introduced in section 1.4.3. They are restated below to simplify matters.

**Req 3.1** Product specification matching should only be based on a given set of product specifications as well as an adequately modeled ontology.

**Req 3.2** A limited set of domain knowledge, e.g., in the form of concept or property synonyms should suffice to execute the matching.

**Req 3.3** The matching result should consist of a set of 1-to-1 mappings with high similarity values.

The input of the algorithms presented in the following is a set of raw product specifications $S_r(x)$ for a product $x$ (step zero in Figure 5.2) being provided by the extraction procedure described in the last chapter. Necessary steps to be taken in the following are pictured in Figure 5.2.

In a first step, the given product has to be categorized using $x$, $S_r(x)$, and an ontology $o$ representing the product information knowledge base. Having the category $c$ at hand, a corresponding subset of ontology elements can be chosen from $o$ to execute the matching process with $S_r(x)$ in step two. The result is a set of matched product specifications $S_m(x,c)$. Finally, these specifications are to be normalized concerning their values which results in the normalized set $S_n(x,c)$ in step three. This is important since it is not to be expected that all producers represent the specifications’ values in the same format.

The different algorithms are based on a central knowledge model that is used to represent the target terminology. Therefore, section 5.1 first introduces an ontology being adequate for managing product specifications retrieved from the Web before the actual categorization (section 5.2), matching (section 5.3), and normalization (section 5.4) processes are described. The final section provides some details on effective product comparisons being enabled through the matching process. The results of this chapter have been published in [177] (see also [104]).

Again, pseudo code is provided for some procedures to give an idea of how they may be implemented. The syntax is defined in appendix A.1.
5.1. Product Representation

The previous section mentioned the ontology $\mathcal{O}$ as the target model for being able to match and normalize extracted product specifications as well as to represent these specifications in an adequate manner. As described in section 2.3.1 of the basics chapter, it is good practice to divide such a knowledge base into several layers, thereby accounting for the different levels of abstractness. Hence, $\mathcal{O}$ is split up to separate between a domain ontology $\mathcal{O}_d$ (or meta-model) and a specific application ontology $\mathcal{O}_a$ (the actual model). In the following, both ontologies will be outlined. Upper ontologies are not taken into account.

5.1.1. Domain Product Ontology

The basics chapter introduced some domain ontologies which focus on the products section while maintaining a certain level of reusability. These ontologies include GoodRelations [91, 92], eClassOWL [88], and others which definitely provide a robust grounding when modeling product information semantically. However, as depicted in section 3.3, they do not allocate comprehensive sets of product property descriptions which is necessary for effectively matching extracted specifications. Therefore, a new domain ontology $\mathcal{O}_d$ has been developed for the matching process that consists of several concepts with attributes and relations, the most important of which are presented in Figure 5.3.

The main concepts of $\mathcal{O}_d$ are Product and Property. A product is an abstract product description with a name (e.g., “Digital Camera”), a list of synonyms, and a set of Translations being bound to certain Languages (e.g., “Digitalkamera” in German). It may also have a parent product (e.g., “Camera”) and a set of properties that are known for this product type. Like the product, a property has a name (e.g., “Optical Zoom”), translations for that name (e.g., “Optischer Zoom” in German), and possibly a parent property (e.g., “Zoom”). Additionally, properties are described by synonyms, keywords, and a field determining the property’s value as to be clusterable or not. E.g., the property MaximumOpticalZoom is clusterable since its values may be organized in meaningful groups like “5-8”, “8-11”, and “11-15”. Properties also have a PropertyStructure, a PropertyDataType, and a Unit. Property structures describe the format of the property’s value that may be a vector (e.g., “20 x 30 x 10”), a simple scalar (e.g., “automatic”), a range (e.g., “20-40”), or a list (e.g., “10, 20, 25”). The property’s value data type can be boolean, float, integer, or string. A property’s unit is one of several units belonging to the same UnitDomain. For example, metres, miles, and inches would belong to the distances domain.

All mentioned ontology contents are used during the matching step to be described in the next section. Additional concepts are required for being able to normalize matched properties in a last step. Those include PropertyConversions and PropertyConverters. A property conversion holds a set of converters for each property that describes how properties can be split up into more simple properties (e.g., maximum and minimum values) or how a property’s value can be distilled from useless information. Four example converters are included in the diagram. Converters point to dedicated code snippets for executing the conversion process.
Figure 5.3.: Product Domain Ontology.
The designed domain ontology is certainly customized for the matching scenario to be described in the following sections. However, all contained concepts are reusable since they model product information in general without major constraints. Even the characteristics of product properties have been chosen generically and may be reused in other systems.

Having a feasible domain ontology at hand, the application ontology can be developed that represents the model for concrete product instances. This ontology \( o_d \) is described in the next section.

### 5.1.2. Application Product Ontology

Application ontologies are very specific for the scenario they are applied to. Therefore, their reusability is quite low, virtually negligible. The application ontology \( o_a \) for the product specifications scenario consists of concrete product types and corresponding property types. Like mentioned above, the domain ontology \( o_d \) could also be seen as a meta-model for \( o_a \) since a concept \( DigitalCamera \) from \( o_a \) instantiates the concept \( Product \) in \( o_d \). A small excerpt of \( o_a \) is shown in Figure 5.4. It includes corresponding instances as well.

![Image](image.png)

**Figure 5.4.: Excerpt of the Product Application Ontology with Instances.**

As can be seen, a concept \( DigitalCamera \) might have properties such as \( ImageResolution \) or \( ImageSizes \). Each of the properties defined for \( DigitalCamera \) is qualified by the corresponding name, synonyms, keywords, the structure, a data type, and a tag defining the property as to be or not to be clusterable. In the case of \( ImageResolution \), the name would be “Image Resolution”, the synonyms might include “Effective Pixels”, the keywords could contain “Approx.”, the structure is scalar, the data type float, and the clusterable tag true.

Both described ontologies were modeled in OWL and build the TBox of \( o \). The ABox is to be filled by the product specifications matching algorithm based on a set of product specifications being extracted from producer websites. As extending the TBox is a laborious task, the following section gives an outlook on how this problem could be solved.
5.1.3. Product Ontology Management

Product ontology management is a wide field including techniques for manually extending ontologies based on certain principles as well as supporting an ontology engineer by automatically suggesting new ontology elements in a limited manner. Both techniques are addressed briefly in the following.

Manual Product Ontology Management

During the last years, a lot of different ontology modeling tools have been developed, examples of which are Protégé [128] and OntoStudio [135]. Such tools offer great help in visually designing an ontology, thus enabling people not knowing about ontology serialization formats to create actual ontologies, e.g., in OWL/XML. However, if thinking about potential product consumers that like to compare products with each other based on an internal knowledge representation, modeling this information with Protégé would still be too challenging. Therefore, an approach for visually extending the product ontology presented above has been developed. It is based on a visualization tool written in JavaScript and hides all the complexity of the internal model from the user by only focusing on the most important concepts. A step-wise navigation through the different product types as well as available product properties helps the user to find the right location where a concept might be missing or needs to be changed for enabling a desired product comparison feature. This approach is certainly quite application-specific. However, if an ontology should be managed in a Wiki-like manner, it offers a feasible solution for this problem. The visualization is detailed in the implementation section of chapter 6.

Automatic Product Ontology Management

Extending a product ontology manually is a cumbersome task. This is even true for the visual editing mentioned in the previous section if the ontology engineer wants to add a great amount of new concepts. Thus, ontology learning mechanisms would be of great help. Such mechanisms are subject to current research works [35, 147]. While many ontology learning approaches focus on extending a knowledge base through information given on the Web in general, the system presented here offers alternative starting points.

The extraction algorithms from section 4.2.2 are mostly independent of the internal knowledge model. Thus, they might find product specifications, even if they are not modeled in the current ontology. Consequently, such product specifications cannot be matched with any of the concepts of \( o \). However, all such product specifications can be saved as potential property candidates and reused in an ontology learning algorithm for offering new ontology elements to a user. Techniques for creating such suggestions include clustering mechanisms being based on a vector space model. The details of this process are not to be presented here.

With the ontology \( o \) at hand, extracted product specifications are to be integrated with each other. Before this matching step is executed, the category of the corresponding product needs to be detected. This categorization step is presented in the next section.
5.2. Product Categorization

The domain ontology $o_d$ has been presented in detail in the last section. More or less all concepts contained in this ontology are shown in Figure 5.3. It is considered to be complete for its task and should generally not be extended by the matching system or by an expert. The application ontology $o_a$, however, has only been presented exemplary since its content may evolve over time. Users of the matching system might add product types or property types, modify the existing concepts, or delete obsolete elements. Thus, $o_a$ contains thousands of elements after some time period, comprehensively defining the product domain with a special focus on technical product specifications.

The basics chapter showed that it is common practice for matching algorithms to generate similarity matrices that contain pairwise similarity values for each element of both given schemas. For the matching task described in this chapter, one schema would implicitly be given by the extracted product specifications while the other one is the ontology $o$ including $o_a$. Matching a set of extracted specifications with some thousand property types being defined in the ontology would create an extreme overhead concerning the calculations to be made as well as the size of the similarity matrix to be handled. Therefore, the set of target properties needs to be narrowed in advance. This can be achieved by using the categorization mechanism presented in section 4.1.1.

Since the matching of product specifications heavily relies on an available product category and the categorization by public vendor Web Services might fail in some cases, a small extension is presented in this section. Therefore, if the categorization fails, different search engines are used to detect a category with the help of pointwise mutual information [164, 57]. The formula for calculating the pointwise mutual information $PMI$ of two variables $x$ and $c$ is given below (Formula 2.1).

$$PMI(x, c) = \log \frac{P(x, c)}{P(x) \cdot P(c)}$$

The idea is to identify the degree of relationship of a product $x$ and a category $c$ by querying the search engines for the product and the category separately as well as executing a search query for the conjunction of both. By the help of the $PMI$ value, the amount of search hits $P(x, c)$ for the conjunction query can be put into perspective to the separate queries for $x$ and $c$. That is, the less search hits are returned for $x$ and $c$ separately and the more hits are returned for the conjunction of both, the higher is the degree of their relationship.

By executing this process for each category, the most similar product category can be detected for $x$. Since a great amount of product categories could cause a reasonable amount of queries to be made, the approach can be optimized by hierarchically trying out the ontology’s categories and only proceeding with the sub-categories of the most promising parent categories.

Having identified relevant product types and corresponding property types, the actual matching step is to be executed. Details are given in the next section.
5.3. Product Specifications Matching

With the global ontology $o$ for representing the product information terminology at hand, it is now the challenge of this section to present an approach for integrating a given product specifications set of a local producer terminology with this global ontology (the matching type is local $\leftrightarrow$ global, see section 2.3.2). Thus, the aim of the matching algorithm is to calculate $S_m(x, c) = \text{match}(S_r(x), o(c))$, where $S_m(x, c)$ represents the product specifications in the target ontology’s terminology and $S_r(x)$ consists of the same specifications in the producer’s terminology. The following sections will guide through the whole process of applying the matching approach, the first of which gives an overview of the general procedure.

5.3.1. General Procedure

The objective of the matching procedure is to create integrated sets of product specifications originally gathered on the Web for being able to compare products effectively. Therefore, in the following a set of elementary matchers as well as an evolutionary matcher and a Naïve Bayes matcher combining the results of the elementary matchers are developed. Like in COMA [48], both combining matchers are executing the elementary ones in parallel. Hence, they belong to the class of composite matchers. During the adoption of the evolutionary matcher, various thresholds and weights are learned to select the most robust alignments. The overall process is presented in Figure 5.5.

As can be seen in the figure, a set of raw technical specifications $S_r(x)$ (e.g., “Number of effective pixels: 10.75 megapixels”) of a product $x$ from a dedicated web page as well as the relevant part of the ontology determined by the product’s category are given as input for the matching system. Each extracted specification is then split up into its key and value. The key is provided to the name matcher that compares it with potential matching properties’ names. The value is provided to four additional matchers, namely a keyword matcher, a structure matcher, a data type matcher, and a unit matcher. Each of these value matchers accesses some knowledge being retrieved from the ontology to generate a similarity value. The result set consists of several similarity matrices where each field describes the matching probability of an extracted specification $s_r$ and a property $p$. Although the figure only shows one similarity matrix per matcher, some of the basic matchers may produce several similarity matrices. These matrices are then provided to an evolutionary matcher as well as a Naïve Bayes matcher. Both matchers calculate composite similarity values for pairs of product specifications and properties taking all given matrices into account. The results are aggregated in one similarity matrix. Finally, the best matching pairs are considered to constitute the set of matched product specifications $S_m(x, c)$ in a final selection step.

In the following, each of the mentioned tasks is inspected in detail, the first of which is the element-level matching.
Figure 5.5.: Ontology Matching of Product Specifications.


5.3.2. Elementary Matchers

As mentioned above, five different elementary matcher types have been developed to solve the allocation problem of extracted product specifications and modeled product property types. All of these matchers are element-based. Unfortunately, extracted product specifications do not offer any schema hierarchy that might be exploited during the matching process, thus preventing the use of structure-level matchers as employed in Similarity Flooding [126]. Each matcher type is defined through a similarity function $\Gamma$ calculating the similarity of a product specification $s_r$ and a property $p$ for the respective matching scenario. Furthermore, all matchers are categorized taking the matcher taxonomy of Shvaiko et al. [157] into account.

Since the four elementary matchers focusing on a specification’s value are based on an extraction function, it is briefly described here. The extraction function is defined by $E : val(s_r), pat \mapsto val(s'_r), val(s'_r) \subseteq val(s_r)$. It is able to detect certain patterns in a given string $val(s_r)$ representing the value of a product specification $s_r$. Therefore, regular expressions are employed to find corresponding matches. The extraction function then returns the longest value part of the string that matches the provided pattern. An example is the regular expression /\d+\d+\s*\d+/i being applied on “4,288 x 2,848 including package”. The extraction function $E$ would yield the substring “4,288 x 2,848” in this case.

The first matcher to be presented is the name matcher which executes string comparisons.

Name Matcher

When it comes to matching elements from different schemas, the most basic way to compare them is by operating on the elements’ names and calculating a corresponding string distance. Thus, a name matcher has been developed that produces several similarity matrices based on the key of a given specification $s_r$ and the name of a property $p$. The resulting similarities are denoted by $\Gamma_{ident}(s_r, p), \Gamma_{contain}(s_r, p), \Gamma_{wleven}(s_r, p), \Gamma_{cleven}(s_r, p), \Gamma_{wgram}(s_r, p), \Gamma_{cgram}(s_r, p)$. In the following, first the general string similarities $\Gamma_{str}$ are defined before they are enhanced to match actual product specifications with properties.

An obvious comparison is to check whether both given strings are identical. A value of 1 is used as similarity value in that case. If the given strings are not identical, the similarity is set to 0. Thus, the identity similarity $\Gamma_{strident}(str_1, str_2)$ is calculated by Formula 3.2.

$$\Gamma_{strident}(str_1, str_2) = \begin{cases} 1 & \text{if } str_1 = str_2 \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

The containment similarity checks if string $str_1$ is contained in string $str_2$ or vice versa and calculates the ratio of their lengths. Formula 3.3 presents details for $\Gamma_{strcontain}(str_1, str_2)$. 

If the strings are not identical and no string contains the other one, the Levenshtein distance \( \delta \) can be calculated for describing the matching similarity. The result of \( \delta \) is absolute since it counts the number of necessary insertions, deletions, and substitutions for transforming a string into another one. Hence, it is common to divide the Levenshtein distance by the length of the longer string to turn it into a relative measure. After subtracting this relative distance from 1, it describes the similarity of two given strings. The resulting formula for \( \Gamma_{str\text{leven}}(str_1, str_2) \) is shown in the following (Formula 3.4).

\[
\Gamma_{str\text{leven}}(str_1, str_2) = 1 - \frac{\delta(str_1, str_2)}{\max\{||str_1||, ||str_2||\}}
\] (3.4)

\( \Gamma_{str\text{leven}}(str_1, str_2) \) produces a similarity of 1 if both strings are identical and detects containments as well. It could thus replace the string similarities presented before. However, for being able to provide as many features as possible to the composite matchers, all name similarities will be kept separated in this way. Additionally, the Levenshtein distance can be calculated on word basis or character basis. Therefore, two different Levenshtein similarities, namely, \( \Gamma_{str\text{wleven}}(str_1, str_2) \) and \( \Gamma_{str\text{cleven}}(str_1, str_2) \) will be used during the matching task.

Product specification keys and property names are the most significant characteristics when searching for similarities between two schema elements. Thus, elevated concepts like n-grams are to be adopted as well. An n-gram consists of a subsequence of items being extracted from a string. Depending on the granularity level, items can be characters, syllables, words, or any other subsequence being sustained by partitioning the string according to a defined criterion. For the name matcher, word-level n-grams and character-level n-grams have been chosen. The word-level 2-grams for a string “Picture Control System” consist of “picture control” and “control system”. Correspondingly, character-level 5-grams include “pictu”, “contr”, or “ontro”. Character-level n-grams containing whitespaces are ignored. Moreover, n-grams are lowercased for easing comparisons.

The function for retrieving all n-grams of an arbitrary level for a string \( str \) is denoted by \( ngrams(str, n) \). Since the amount of similarity matrices might massively increase if one matrix was created for each \( n \), the concept of n-to-m-grams is introduced. The corresponding formula is presented in the following (3.5).

\[
ntomgrams(str, n, m) = \prod_{i=n}^{m} ngrams(str, i)
\] (3.5)

As can be seen, the set of n-to-m-grams consists of all n-grams for a string \( str \) that have a length of \( n \) up to \( m \). For example, the set of character-level 2-to-8-grams could be
calculated for the product specification key mentioned above. With such sets at hand, an overall similarity $\Gamma_{str_{tonogram}}(str_1, str_2, n, m)$ can be designed based on the Jaccard index (Formula 3.6).

$$\Gamma_{str_{tonogram}}(str_1, str_2, n, m) = \frac{|tonograms(str_1, n, m) \cap tonograms(str_2, n, m)|}{|tonograms(str_1, n, m) \cup tonograms(str_2, n, m)|}$$ (3.6)

The formula shows that the amount of n-to-m-grams located in both given strings is normalized by the total amount of unique n-to-m-grams. Finally, the similarities $\Gamma_{<n>to<m>wgram}(str_1, str_2)$ and $\Gamma_{<n>to<m>cgram}(str_1, str_2)$ are calculated based on $\Gamma_{str_{tonogram}}(str_1, str_2, n, m)$ by using words or characters, respectively. $\langle n \rangle$ and $\langle m \rangle$ are place holders for concrete length values of the n-grams.

The different similarity measures explore the product specification keys and property names for finding similarities. However, a further promising extension is to take alignments from previous matching tasks into account to find potential transitive mappings. All product specification keys of successful previous matching tasks are contained in the corresponding property’s synonym set. The final name similarity calculation is shown in Formula 3.7. $\Gamma_{<name>}(s_r, p)$ is a place holder for the extended similarity functions presented above, that is, $\Gamma_{ident}(s_r, p)$, $\Gamma_{contain}(s_r, p)$, $\Gamma_{wleven}(s_r, p)$, $\Gamma_{cleven}(s_r, p)$, $\Gamma_{<n>to<m>wgram}(s_r, p)$, and $\Gamma_{<n>to<m>cgram}(s_r, p)$.

$$\Gamma_{<name>}(s_r, p) = \max(\{\Gamma_{str_{<name>}}(str_1, str_2)|str_1 = key(s_r), str_2 \in name(p) \cup syn(p)\})$$ (3.7)

For the strings “Effective Pixels” and “Pixels”, the name matcher would calculate the case-insensitive similarities $\Gamma_{ident}(s_r, p) = 0$, $\Gamma_{contain}(s_r, p) = 0.375$, $\Gamma_{wleven}(s_r, p) = 0.5$, $\Gamma_{cleven}(s_r, p) = 0.375$, $\Gamma_{1to2wgram}(s_r, p) = 0.3$, and $\Gamma_{2to8cgram}(s_r, p) = 0.3$. If a synonym of “Pixels” would be “Resolution”, “Pixels” would still be chosen as matching partner since it produces higher similarity values when being compared to “Effective Pixels”.

Concerning the matcher categories introduced in section 2.3.2, the name matcher is string-based and alignment reusing. Especially the alignment reusing feature, like it was first introduced in COMA, is of great importance since it adds a learning component to the name matcher. Hence, with a growing set of successful matchings, the name matcher produces better results.

In the following, the extracted specifications’ values will be examined. The first of four developed value matchers focuses on keywords contained in a value.

**Keyword Matcher**

In many cases product specification values do not include numerical or boolean values but describe a product’s feature by textual means, e.g., the types of supported memory cards by “SD, SDHC”. Therefore, a keyword matcher has been developed that calculates links to properties with the help of the keyword similarity $\Gamma_{key}(s_r, p)$.

The keyword matcher is based on the initially mentioned extraction function $E(val(s_r), pat)$. For this matcher, the extraction patterns to be used consist of the set of known keywords.
per property. Since specification values, and therefore identified keyword matches, can be of arbitrary length, the length of the identified pattern has to be normalized by the complete value length. Additionally, the relative pattern matching value per keyword is summed up for all keywords of a property \( p \) to calculate the overall keyword similarity of a product specification \( s_r \) and a property \( p \). Formula 3.8 details the calculation.

\[
\Gamma_{\text{key}}(s_r, p) = \sum_{\text{pat} \in \text{keywords}(p)} \frac{|E(\text{val}(s_r), \text{pat})|}{|\text{val}(s_r)|}
\] (3.8)

For a given value “Up to 512 MB: SD, MMC” and a property “Supported Memory Cards” with keywords “MMC” and “RS-MMC”, the keyword matcher calculates the similarity \( \Gamma_{\text{key}}(s_r, p) = 0.143 \).

As the keyword similarity approximation is based on a dictionary of keywords per property, the keyword matcher belongs to the class of linguistic resources matchers. Similarly to the name matcher, it is a learning matcher since unknown strings being detected in a correctly matched product specification value may be reused for solving future matching tasks. The following matcher focuses on structures.

**Structure Matcher**

The structure of a product specification’s value may provide additional hints on which properties are to be matched with the product specification of interest. It has been identified to assume one of four shapes, namely, a list, a range, a vector or a scalar. A list of atomic values is typical for product specifications such as “Supported Flash Cards: SD, MMC”. Ranges of atomic values look like “2x - 12x” for specifications such as “optical zoom”. Vector values are often used for attributes like “Size: 20cm x 10cm x 15cm”. The basic value structure is scalar, that is, the value itself is already atomic. If a product specification value’s structure can be successfully detected, it enables the identification of potential matching properties.

For being able to calculate the structural similarity of a product specification \( s_r \) and a property \( p \)’s structure defined in the ontology \( o \), again an extraction function \( E : \text{val}(s_r), \text{pat} \) is adopted that searches for patterns of all the mentioned shapes in a given specification value and extracts the longest string complying with such a pattern for each structure type. As for the keyword matcher, the found pattern matches are normalized by the value length. Formula 3.9 shows the resulting function \( \Gamma_{\text{struct}}(s_r, p) \).

\[
\Gamma_{\text{struct}}(s_r, p) = \max_{\text{pat} \in \text{patterns(struct(p))}} \left( \frac{|E(\text{val}(s_r), \text{pat})|}{|\text{val}(s_r)|} \right)
\] (3.9)

For a given value “9..99 years” and a property “Age” with the range structure, the structure matcher would calculate the similarity \( \Gamma_{\text{struct}}(s_r, p) = 0.455 \). A proper pattern to be used in this case is the regular expression \(/([\backslash d,\backslash] + \backslash s \ast \backslash \backslash \backslash s \ast ([\backslash d,\backslash] +))/i\).

The structure matcher implementing the presented similarity measure belongs to the constraint-based class since it exploits a special feature of the product specification’s type. The following data type matcher is to be located in a similar category.
Data Type Matcher

The data type matcher is the third value matcher type presented here and calculates the similarity between a product specification value’s data types and a set of defined property data types in the ontology $o$ by the use of $\Gamma_{\text{type}}(s_r, p)$. Three specific data types have been identified, namely, boolean, float, and integer. The data type string is used as a general standard type.

The calculation of the similarity is analog to the one of the structure matcher. Again, by the use of an extraction function $E(val(s_r), pat)$, data type patterns are located in the product specification value. The similarity function $\Gamma_{\text{type}}(s_r, p)$ normalizes the length of the found pattern by the overall value length. Formula 3.10 provides more details.

$$\Gamma_{\text{type}}(s_r, p) = \max_{\text{pat} \in \text{patterns}(\text{type}(p))} \left( \frac{|E(val(s_r), \text{pat})|}{|\text{val}(s_r)|} \right)$$

For the string type no patterns are available since it is not specific enough. An example of a calculated data type similarity would be $\Gamma_{\text{type}}(s_r, p) = 1$ concerning the value “yes” and a property “Supports JPG” with the boolean data type.

As mentioned in the last section, the data type matcher is similar to the structure matcher and is thus also categorized as a constraint-based matcher. The last developed matcher is the unit matcher which is to be presented in the following section.

Unit Matcher

The unit matcher calculates the similarity $\Gamma_{\text{unit}}(s_r, p)$ which is based on the detection of measurement units in given product specification values. The basic idea is similar to the previous value matchers. An extraction function $E(val(s_r), pat)$ uses patterns for identifying units in the specification values. The normalization of the similarity value is done by the specification value length.

Of the vast amount of measurement units being employed in the technical product domain, some units can be categorized to be more specific for a product property while others are quite generic and may occur in many different properties. It is therefore valuable to take a unit’s specificity $\Upsilon(u)$ into account since highly specialized units that are utilized in only very few cases provide a much stronger link between a product specification and a property. A unit’s specificity depends on how many properties in $o_a$ are connected to this unit. The more properties use a particular unit, the less specific the unit is for each of these properties (Formula 3.11).

$$\Upsilon(u) = \frac{1}{|\{q | q \in P \land u = \text{unit}(q)\}|}$$

By combining the formulas of the previous matchers with the introduced specificity $\Upsilon(u)$, the overall unit similarity can be calculated as shown below (Formula 3.12).

$$\Gamma_{\text{unit}}(s_r, p) = \Upsilon(\text{unit}(p)) \cdot \max_{\text{pat} \in \text{patterns}(\text{unit}(p))} \left( \frac{|E(val(s_r), \text{pat})|}{|\text{val}(s_r)|} \right)$$
Adopted measurement units depend on preferences of the presenting producer as well as local peculiarities for the region the producing company is located in. Thus, two different producers might use varying units for describing product specifications being related to the same product property. For example, one producer might choose centimetres to determine the size of a product while another producer prefers inches. It is therefore beneficial to provide a unit model to the matcher that enables the identification of units relating to the same unit domain. Both, units and unit domains have been included in the domain ontology presented above. An excerpt of the developed unit model is shown in Figure 5.6. The relations between units of the same domain allow a wider search for unit patterns. The connections between the unit types are marked with a float value enabling the conversion between each other.

![Figure 5.6.: Example Unit Model for the Distances Domain.](image-url)

When provided with such a unit model, a respective function can detect a relevant unit domain by searching for all its units in a specification’s value. The domain similarity $\Gamma_{\text{dom}}(s_r, p)$ is calculated by summing up all similarities for identified units as shown in Formula 3.13.

$$
\Gamma_{\text{dom}}(s_r, p) = \sum_{u \in \text{units}(\text{dom}(p))} \Upsilon(u) \cdot \max_{\text{pat} \in \text{patterns}(u)} \left( \left| \frac{E(\text{val}(s_r), \text{pat})}{|\text{val}(s_r)|} \right| \right)
$$

Consequently, the unit similarity of a given value “8 Megapixels (maximum)” and a property “Video Effective Pixels” belonging to the resolutions domain would result in $\Gamma_{\text{unit}}(s_r, p) = \Upsilon(“megapixels”) \cdot 0.455$. Concerning the matcher categorization in section 2.3.2, the unit similarity matcher belongs to the class of linguistic resources matchers since it uses a kind of lexicon to enable the matching process. The lexicon consists of all unit models included in the ontology.

Having a set of element-level matchers at hand, two composite matchers can be developed that process the created similarity matrices of defined properties $P$ and extracted specifications $S_r$ to calculate a composite one. The first composite matcher is based on various weights and thresholds that are to be learned by an evolutionary algorithm. The second composite matcher applies Naïve Bayes. Both are presented in the next two sections.
5.3.3. Evolutionary Matcher

Evolutionary algorithms optimize problem solutions by imitating techniques of natural evolution. This includes trying out different mutations of a given configuration to generate the best possible output. For the matching task presented in this section, the idea is to adapt a set of weights and thresholds to create the optimal set of matching pairs.

Therefore, the previously defined similarity measures first need to be extended by a threshold \( \tau \) and a weight \( \omega \). Both variables have a value between zero and one. \( \tau \) is applied by a threshold function \( \Theta_\tau \) that is introduced in Formula 3.14.

\[
\Theta_\tau (v) = \begin{cases} 
  v & \text{if } v \geq \tau \\
  0 & \text{otherwise}
\end{cases}
\]  

(3.14)

As can be seen, \( \Theta_\tau \) returns zero if the given value does not exceed a certain threshold \( \tau \). The weight \( \omega \) determines the significance of a similarity function. The composite similarity measure of the evolutionary matcher is calculated by a formula that aggregates the different previously presented similarity functions and applies corresponding weights and thresholds (Formula 3.15).

\[
\Gamma_{\text{spec}}(s_r, p) = \Theta_{\text{spec}} \left( \sum_{\text{sim} \in \text{similarities}} \omega_{\text{sim}} \cdot \Theta_{\text{sim}} \left( \Gamma_{\text{sim}}(s_r, p) \right) \right) / |\text{similarities}|
\]  

(3.15)

The sum of all similarities is divided by the number of calculated similarities to obtain the arithmetic mean. Finally, the function \( \Theta_{\text{spec}} \) is applied. Thus, a potential match is only accepted if the overall specification similarity is above \( \tau_{\text{spec}} \). It is to be emphasized that all previously calculated similarities have a value between 0 and 1. Hence, also the composite similarity returns a value in this range. The arithmetic mean could neither be replaced by the geometric mean, nor by the harmonic one since both return 0 if at least one factor is 0.

Originally, a decision tree-based matching process had been developed inspired by MatchPlanner [50]. A preliminary evaluation however proved that the negligible amelioration of matching results does not justify accruing additional expenses and decreasing flexibility concerning the addition and removal of similarity measures.

The resulting matrix offers similarity values for each pair of product specifications and properties. The next section shows how to learn included thresholds and weights.

Evolutionary Learning

When calculating similarity values for given pairs of product specifications and properties, a pre-selection has to take place that dismisses pairs with a similarity value below a certain threshold. Additionally, some matching components deliver valuable results with a higher probability. Thus, weights need to be employed that give more importance to more reliable matching techniques. An overview of all introduced thresholds \( \tau \) and weights \( \omega \) is given in Table 5.1.

As mentioned above, the composite matcher can be provided with initial, manually created values for the different thresholds and weights. However, optimizing a matching
Table 5.1.: Thresholds and Weights of the Composite Matcher.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_{\text{spec}})</td>
<td>Threshold</td>
<td>0.06</td>
<td>Minimum Overall Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{spec}})</td>
<td>Weight</td>
<td>1</td>
<td>Rates Overall Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{ident}})</td>
<td>Threshold</td>
<td>1</td>
<td>Minimum String Identity Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{ident}})</td>
<td>Weight</td>
<td>0.98</td>
<td>Rates Identic String Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{contain}})</td>
<td>Threshold</td>
<td>0.06</td>
<td>Minimum String Containment Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{contain}})</td>
<td>Weight</td>
<td>0.94</td>
<td>Rates Contained String Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{ntomwgram}})</td>
<td>Threshold</td>
<td>0.05</td>
<td>Minimum Word-Level n-to-m-Gram Sim.</td>
</tr>
<tr>
<td>(\omega_{\text{ntomwgram}})</td>
<td>Weight</td>
<td>0.92</td>
<td>Rates Word-Level n-to-m-Gram Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{ntomegram}})</td>
<td>Threshold</td>
<td>0.10</td>
<td>Minimum Char-Level n-to-m-Gram Sim.</td>
</tr>
<tr>
<td>(\omega_{\text{ntomegram}})</td>
<td>Weight</td>
<td>0.93</td>
<td>Rates Char-Level n-to-m-Gram Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{eleven}})</td>
<td>Threshold</td>
<td>0.11</td>
<td>Minimum Word-Level String Distance Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{eleven}})</td>
<td>Weight</td>
<td>0.92</td>
<td>Rates Word-Level String Distance Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{cleven}})</td>
<td>Threshold</td>
<td>0.4</td>
<td>Minimum Char-Level String Distance Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{cleven}})</td>
<td>Weight</td>
<td>0.92</td>
<td>Rates Char-Level String Distance Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{struct}})</td>
<td>Threshold</td>
<td>0.01</td>
<td>Minimum Value Structure Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{struct}})</td>
<td>Weight</td>
<td>0.49</td>
<td>Rates Value Structure Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{type}})</td>
<td>Threshold</td>
<td>0</td>
<td>Minimum Value Data Type Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{type}})</td>
<td>Weight</td>
<td>0.69</td>
<td>Rates Value Data Type Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{unit}})</td>
<td>Threshold</td>
<td>0</td>
<td>Minimum Value Unit Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{unit}})</td>
<td>Weight</td>
<td>0.95</td>
<td>Rates Value Unit Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{dom}})</td>
<td>Threshold</td>
<td>0</td>
<td>Minimum Value Unit Domain Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{dom}})</td>
<td>Weight</td>
<td>0.9</td>
<td>Rates Value Unit Domain Similarity</td>
</tr>
<tr>
<td>(\tau_{\text{key}})</td>
<td>Threshold</td>
<td>0</td>
<td>Minimum Values Keywords Similarity</td>
</tr>
<tr>
<td>(\omega_{\text{key}})</td>
<td>Weight</td>
<td>0.78</td>
<td>Rates Values Keywords Similarity</td>
</tr>
</tbody>
</table>
system manually is time-consuming. Hence, the detection of such values should be automatized by the help of a so-called gold standard. A valuable gold standard for the matching task offers extracted product specifications as well as links to corresponding product properties. Using a training set being part of this gold standard, each matcher can be optimized separately by incrementing or decrementing the defined thresholds and weights and measuring if the matching results improved for this change or not, iteratively. Typical thresholds and weights which have been calculated based on such a gold standard are also shown in Table 5.1. Details on gold standard creation are given in section 6.2.2.

The rule of thumb is to first detect the optimal thresholds for each elementary matcher and then assign corresponding weights. With the basic matcher configurations at hand, the overall threshold $\tau_{\text{spec}}$ can be calculated. An optimal threshold is defined as the threshold leading to matching results with the maximum $F_1$ score. The weight is provided by the precision value belonging to the calculated $F_1$ score. Details on the detection of an optimal threshold are given in Algorithm 5.1.

**Algorithm 5.1** Evolutionary Threshold Calculation.

```ruby
1 # The calculation is based on the accuracy of the thresholds t_accuracy and
2 # frequency of thresholds t_frequency per interval.
3 t_accuracy = 3
4 t_frequency = 10
5 best_t = 0
6 left_best_t = 0
7 right_best_t = 1
8
9 t_accuracy.times do
10   # Calculate all F1 scores with corresponding thresholds for the current
11   # interval.
12   f1_t = {}
13   interval_size = right_best_t - left_best_t
14   (t_frequency + 1).times do |t_index|
15     t = left_best_t + interval_size/t_frequency * t_index
16     f1_t[calculate_f1_score(t)] = t
17   end
18
19   # Detect the threshold with the highest F1 score as well as a local
20   # maximum left of best_t and a local maximum right of best_t as borders
21   # for the new interval.
22   best_t = f1_t[f1_t.keys.max]
23   best_t_index = f1_t.values.index(best_t)
24   left_best_t = f1_t[f1_t.keys[0, best_t_index].max]
25   right_best_t = f1_t[f1_t.keys[best_t_index + 1, f1_t.values.length].max]
26 end
27
28 # The best threshold is returned. For example, for f1_t = {0.786 => 0.2,
29 # 0.733 => 0.3, 0.654 => 0.4}, the threshold best_t = 0.2 would be returned.
30 return best_t
```
As can be seen in line 16, the algorithm makes use of the $F_1$ score to detect an optimal threshold. Lines 3 to 7 contain some initializations. $t_{\text{accuracy}}$ defines how many iterations are to be made in the evolutionary algorithm. $t_{\text{frequency}}$ determines how many potential thresholds are to be examined during each iteration. $\text{best}_t$ is the optimal threshold while $\text{left}_t$ and $\text{right}_t$ describe the optimal thresholds left and right of $\text{best}_t$, respectively. In line 12, the map containing $F_1$ scores and thresholds $f1_t$ is defined which is filled by corresponding values in line 16. In lines 22 to 25 $\text{best}_t$, $\text{left}_t$, and $\text{right}_t$ are calculated for each iteration. Finally, $\text{best}_t$ is returned in line 30. The algorithm is first executed for each elementary matcher separately and then for the overall matcher.

Depending on the reliability the thresholds need to achieve, the initial values for $t_{\text{accuracy}}$ and $t_{\text{frequency}}$ may be changed. In most cases, the given ones are fully adequate for the evolutionary learning algorithm. It is to be pointed out that the algorithm may not find the maximum of an arbitrary function for low frequency values. However, a graph of the matching task’s $F_1$ score is fairly monotone. Thus, outliers that could be overseen at low sampling frequencies are unusual. Furthermore, since all thresholds and weights for elementary similarities are calculated individually, they are assumed to be independent from each other. This is certainly not true. However, machine learning approaches like the subsequently presented Naïve Bayes classifier actually do not suffer from this assumption making it feasible for the presented learning approach as well.

The strength of the described evolutionary matcher is that it may already be executed with a manual configuration as the amount of parameters to adjust is limited. However, with a gold standard of reasonable size at hand, available machine learning techniques can be applied to execute the matching task. It would therefore be advisable to add another, machine learning-based component like the Naïve Bayes matcher which is described in the next section.

### 5.3.4. Naïve Bayes Matcher

A Naïve Bayes classifier is an algorithm which allots a certain class to each given object using a cost function. It is based on the theorem of Bayes. The corresponding formula is presented below (Formula 3.16).

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$  (3.16)

The theorem allows the calculation of the conditional probability of $A$ if $B$ is given. Naïve Bayes assumes that all attributes providing a link to a certain class are of equal importance and that one attribute’s value does not depend on other attribute values of the same object. As mentioned above, this assumption is generally not true in reality. However, Naïve Bayes creates valuable results in many cases. It is also employed for the matching task. Since Naïve Bayes classifiers are offered by many frameworks for different programming languages and to keep this section short, the internal functionality of such a classifier is not described here.
The input of the Naïve Bayes matcher consists of a set of fields including all calculated basic similarities from the elementary matchers. Additionally, a field called *match* is added that describes if a product specification and a property have been chosen to correspond to each other. Based on a series of such records called *training set*, the classifier learns the different classes of correct matches and, hereafter, detects new matches automatically.

Applying machine learning techniques like Naïve Bayes for matching tasks is typical for approaches employing instance data. Examples are Automatch [19] and GLUE [49] which heavily rely on Naïve Bayes.

The Naïve Bayes matcher includes a learning component for finding the optimal configuration as well. Since the configuration must not be tried out on the test set, the training set has to be divided into another training and test set. The learning component simply tries out all possible combinations of similarities to find the configuration with the best F1 score on these sets. Having the optimal configuration at hand, the actual test set can be matched for calculating the overall F1 score.

With a similarity matrix containing the alignment $S_m(x, c)$ as well as additional false pairs at hand, a final selection has to take place. During this selection step the Stable Marriages Problem needs to be taken into account which is depicted in the following section.

### 5.3.5. Result Selection

Matching extracted product specifications with properties from the ontology by using the composite matchers developed above creates similarity matrices with many potential pairs. If the property set for the detected product type is complete, all correct matches are included in such a similarity matrix. The task is now to eliminate matches that link two elements by mistake. A major amount of such matches has already been filtered out during the matching step itself by applying threshold functions. However, the resulting matrix will still contain elements that match several other elements with different similarity values. An example is shown in Table 5.2.

<table>
<thead>
<tr>
<th></th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{r_1}$</td>
<td>0.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s_{r_2}$</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s_{r_3}$</td>
<td>0.8</td>
<td>0.9</td>
<td>0.8</td>
<td>0</td>
</tr>
<tr>
<td>$s_{r_4}$</td>
<td>0</td>
<td>0</td>
<td>0.7</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The easiest way to select matching pairs would be to choose the strongest links, that is, the matching links with the highest similarity value and to discard the others. As shown by the table, $s_{r_1}$ as well as $s_{r_2}$ could be matched with $p_1$ if choosing the maximum value row-wise. Unfortunately, $p_1$ prefers $s_{r_3}$ as a matching partner. Furthermore, only one-to-one matches are accepted during the selection step since a product specification should always correspond to exactly one property and vice versa. If a product specification resides on a more coarse-grained level and would thus match several properties, an
equally coarse-grained property needs to be selected as matching partner that is to be split up into detailed properties during the subsequent normalization step. Hence, the selection algorithm has to account for the calculated similarity values while only allowing one-to-one matches.

The presented problem is known as the Stable Marriages Problem. It describes a situation of several marriages where each partner has an ordered list of preferences for partners of the opposite sex. The goal is to find a set of marriages such that no two partners of opposite sex prefer themselves over their assigned spouses. A solution for the Stable Marriages Problem is the Gale-Shapley-Algorithm [71]. They proved that it is always possible to solve the Stable Marriages Problem for an equal number of men and women. A similar algorithm has therefore been developed for the presented selection problem. Corresponding pseudo code is shown in Algorithm 5.2.

The algorithm operates in three major steps. In the first step (lines 8 to 17), for all properties (or columns), all similarity values below the maximum value of the current column are set to zero. Before, these values have been saved in the candidates matrix. For the example matrix in Table 5.2, this would mean that 0.7 and 0.5 are deleted in column $p_1$ and 0.7 is deleted in column $p_3$. The second and the third step are executed until only non-ambiguous matchings remain. The second step (lines 19 to 29) iterates through all product specifications (or rows) and sets similarities below the maximum similarity of the corresponding row to zero. Thus, for row $s_{r_3}$, both 0.8 values are removed. Then, in the third step (lines 33 to 44), for each property (or column), its highest similarity value is copied back from the candidates matrix if no similarity value above zero is available anymore. The resulting similarity matrix after having executed the selection step is shown in Table 5.3.

<table>
<thead>
<tr>
<th></th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{r_1}$</td>
<td>0.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$s_{r_2}$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$s_{r_3}$</td>
<td>0.0</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$s_{r_4}$</td>
<td>0.0</td>
<td>0.7</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Since identical similarity values in one row could cause the algorithm to stop, the actual implementation includes a small extension. However, it is very unlikely that calculated similarity values are identical.

Based on an optimized set of one-to-one matches, the product specifications can be cleaned to make them comparable by machines and potential consumers. This step is executed by several product specifications normalization algorithms. The normalization is the final step in the overall process chain and is to be described in the following.
Algorithm 5.2 Finding Stable Marriages.

1. # The similarity matrix contains similarities for pairs of properties
2. # (columns) and product specifications (rows).
3. sim_matrix = match_raw_product_specificatons(raw_specs, properties)
4. cands_matrix = sim_matrix
5. 
6. # For all properties, remove all similarities below the maximum similarity
7. # from the similarity matrix and remember those in the candidates matrix.
8. sim_matrix.each_column_with_index do |column, column_index|
9.   max_sim = column.max
10. column.each_with_index do |sim, row_index|
11.     if(sim == max_sim)
12.         cands_matrix[column_index][row_index] = 0
13.     else
14.         sim_matrix[column_index][row_index] = 0
15.     end
16. end
17. 
18. while(similarity_matrix.contains_rows_with_several_values_above_zero)
19.   # For all product specifications, remove all similarities below the maximum
20.   # similarity from the similarity matrix.
21.   sim_matrix.each_row_with_index do |row, row_index|
22.     max_sim = row.max
23.     row.each_with_index do |sim, column_index|
24.       unless(sim == max_sim)
25.         sim_matrix[column_index][row_index] = 0
26.       end
27.     end
28. end
29. 
30. # For all properties, copy the highest similarity value from the candidates
31. # matrix back to the similarity matrix if no similarity value is above 0.
32. sim_matrix.each_column_with_index do |column, column_index|
33.   if(column.max == 0)
34.     max_cand_sim = cands_matrix.column_at(column_index).max
35.     column.each_with_index do |row_index|
36.       if(cands_matrix[column_index][row_index] == max_cand_sim)
37.         sim_matrix[column_index][row_index] = max_cand_sim
38.         cands_matrix[column_index][row_index] = 0
39.       break
40.     end
41.   end
42. end
43. 
44. end
45. end
5.4. Product Specifications Normalization

If an alignment between an extracted specification $s_r$ and a property $p$ has been detected, the correct key of $s_r$ is the one of $p$. However, if the extracted specifications have complex structures, such as lists, vectors, or ranges, contained values are not easily comparable. They have to be atomized therefore. The contained values also need to be cleaned to gain $S_n(x, c)$ as displayed in step three of Figure 5.2. Both steps are presented in the following.

5.4.1. Product Specifications Atomization

In many cases, producers use different granularity levels for describing product specifications on their web pages. To make products comparable, it is important to always present given specifications on the same detail level. Thus, if a product specification has been matched with a complex property, it is to be split up into several elementary and comparable specifications. This is done for lists, vectors, and ranges by the converters defined in the ontology. The converters point to a dedicated code snippet describing how to divide such structures. The snippets may be generic for the assigned structure or specific for properties in a certain category. For example, the value of “Aperture” could be divided into a minimum and a maximum value. Hereafter, the complex specification can be deleted to avoid redundancy.

5.4.2. Product Specifications Value Normalization

Values of extracted product specifications can be of various formats. In many cases, the extraction routine appended strings that are not part of the actual value. Furthermore, producers might have added information that is not important for the actual specification value. Therefore, a series of four steps is to be executed.

In the first step, the values are cleaned by removing HTML tags. Such information may be included depending on the structure of a producer’s web page and can indeed be helpful, e.g., for detecting the value type to be a list during the matching step. However, after having matched and split the specification, such HTML has to be removed.

During the second step, redundant information is removed from the values. For example, if the presenting authority has added nonrelevant information in brackets, it may be removed here.

The third step consists of distilling numeric values or changing boolean values to “true” and “false”, respectively. This can only be done if the value structure is scalar and such numeric values are contained.

In step four, the unit assigned to the product specification’s property is detected. If the unit is found, the algorithm finishes. If an alternative unit from the same domain is found, it is changed to the current specification’s standard unit followed by a recalculation of the associated numeric value. Unit models like the one in Figure 5.6 are used here again.

The resulting product specifications are categorized, matched, and cleaned, and can finally be presented to a potential user in an appealing way.
5.5. Product Comparison

For their homogeneous format, integrated product specifications are perfectly qualified to perform effective product comparisons. Such comparisons can be based on user interface approaches like faceted search. Faceted search interfaces are rich alternatives to traditional query interfaces. For the product domain, such an interface would offer a facet for each product property type being available in a selected category. Inside a facet, values of corresponding products are clustered into clickable value groups that narrow the product result set down to the set of products fulfilling the selected cluster conditions. By such means, the product set can be limited to a group of two or three products being interesting for the user. For these products, effective product comparisons can take place that compare the most important product properties with each other by qualitative measures. For example, a corresponding search system might inform the current consumer that one digital camera is nearly two times as heavy as another one.

User interfaces are not in the focus of this chapter. Therefore, some more details are provided in the implementation section of the next one. In the following, a conclusion outlines the results of this chapter.

5.6. Conclusions

The current chapter dealt with product information integration for the Web. The central data structure was the one of a product specification originally being retrieved by the extraction mechanisms described in the previous chapter. The goal of this chapter was to develop algorithms for categorizing, matching, normalizing, and managing products and their corresponding product specifications.

The first section introduced a means for representing product information by the use of ontologies. The ontology \( o \) consists of a domain ontology universally describing product information. Central concepts of the ontology include \textit{Product} and \textit{Property}. Several additional concepts especially focus on the comprehensive description of product properties which represent the target of the matching process. The application ontology contains concrete product types as well as corresponding property types. The section concluded with some additional information on product ontology management.

The second part provided some information on product categorization. Since the basic functionality had already been explained in section 4.1.1, only the extension of the categorization algorithm has been presented.

Section three contributed the main part of this chapter and focused on the actual product specifications matching process. It first provided an overview of the general matching procedure which consists of a basic matching step that creates initial similarity matrices for a set of product specifications and potential matching properties, the processing of these matrices in two composite matchers, and a final selection step. The first matching step is executed by five different matcher types focusing on the product specification’s name, keywords being contained in its value, the value’s structure and data type, as well as units being found in the value. The composite matchers consist
Conclusions

of an evolutionary matcher and a Naïve Bayes matcher. Both matchers have learning components while the evolutionary matcher already delivers results with an initial manual configuration and the Naïve Bayes matcher especially works well on large data sets. After having completed the matching, a final selection step considers the Stable Marriages Problem when selecting a set of 1-to-1 matchings being returned as the matched product specifications set.

The fourth section offered some details on product specifications normalization. Matched product specifications are to be split up into atomic product specifications for easing the process of comparing them with each other. Additionally, their values are to be cleaned from unwanted or dispensable information.

The last section closed the chapter with a short overview on how such product specifications could be used to effectively compare products.

Like the extraction chapter, also this one focused on the product domain, especially product specifications being provided in a raw format. However, many aspects of the presented results are still reusable. For example, the product domain ontology presented in the beginning is not exclusively suitable for the described matching task. It might as well be used in other applications that need to model product information. It is abstract enough to avoid limitations for such applications. Theoretically, even the application ontology could be reused. The comprehensive amount of information including product types, property types, units, unit domains, translations, etc. required a lot of work which would be saved by the reusing institution. As the ontology is fully modeled in OWL, its deployment would be trivial. The matching task itself is quite specific. Still, the rigorous exploitation of characteristics for given product specifications might inspire other engineers searching for matching characteristics. When designing a matcher, the detection of valuable characteristics is the hardest part.

With the completion of this chapter, the concept for retrieving, extracting, and matching product information in the form of product specifications has been fully exposed. However, a concept is worthless if it cannot be proven. Hence, the next chapter presents details on the implementation of all presented algorithms in a federated search environment called Fedseeko. Hereafter, it puts a special focus on the evaluation of the concepts based on this implementation.
The first chapter of this thesis introduced the topic of federated product information search and semantic product comparisons on the Web. Besides a description of the current product information research scenario, the aspired manner of how to retrieve product information, goals and non-goals, as well as additional information, it presented a set of theses concerning the amelioration of product information search on the Web. They stated that the retrieval and extraction of high-quality product specifications from producer web pages as well as their integration with a central knowledge base could be mostly automated for enabling effective product comparisons. Based on those assumptions, relevant technologies and related work have been examined in chapter 2. A proposition of how to combine such technologies for building a federated product information system has been given in chapter 3. With all relevant knowledge at hand, the basic concepts could eventually be developed in chapter 4 and 5. These concepts are built on theoretical foundations since available technologies as well as related approaches from the different research domains have been reused when possible.

However, before a developed concept may be judged as feasible, it needs to be evaluated. In this case, the evaluation consists of the implementation of all presented algorithms in a federated product information search architecture (section 6.1) as well as a set of measurements conducted especially in the area of product specifications locating, extraction, and matching (section 6.2). The requirements stated in section 1.4.3 need to be satisfied by the implementation. By the help of the obtained measurements, the theses from section 1.4.2 may finally be supported or refuted. Hence, in the following, the implemented components are to be presented first before the actual measurement results are described.

### 6.1. Implementation

Chapters 4 and 5 presented a series of algorithms for the various tasks to be accomplished when retrieving sets of product specifications. These algorithms are categorized by vendor, producer, and third-party information search. Accordingly, a distributed architecture has been designed and implemented that encapsulates each of the major tasks in its own Web Service. The consolidation of the information being provided by the available services may be carried out by different federated product information search portals. Two such consuming portals have been developed to prove the concept, one of which is Fedseeko, a
Web-based application. The other consumer is an iOS application offering the retrieved product information on mobile Apple devices. Figure 6.1 provides an overview of the different Web Services as well as how they can be tied to a web application such as Fedseeko.

Figure 6.1.: The Prototype’s Federated Product Information System Architecture.

The main resources delivered by the three implemented services are offers, products, and snippets. Offers are retrieved through vendor sources while products are built from basic vendor information and additional technical specifications originating from producers. Snippets are simple information sets from third parties.

The major advantage of creating service-based architectures is that processed information becomes reusable. Product information originally being distributed all over the Web in various representation formats can be offered by Web Services in a structured format that may be further processed by an arbitrary consuming application. Concerning the Web Service type, REST has been preferred over SOAP for two reasons. First of all, it concentrates on resources rather than function calls and is thus more appropriate for representing product information. Furthermore, REST services may offer their contents
in various formats that can also be understood by humans.

The reference implementations have been developed in JRuby [132] on Rails [86] with several Java extensions. Each application has been strictly designed according to the Model-View-Controller pattern [68]. Ruby on Rails makes it easy to deliver data in various formats, thus every service can return requested information in ordinary HTML, in XML, or in JSON. Each of the major components will be presented briefly in the following.

6.1.1. Offers Service

The Offers Service [171] is the first of three central services in the developed federated product information system. It focuses on information from online malls, thus relating to Section 4.1. A screenshot of the service’s web interface is given in Figure 6.2.

As can be seen, different vendors are available to be queried for a set of relevant offers. A consuming application might also request several sources at a time, hence, benefiting
from the included ranking algorithms. A corresponding query URL directed to the Offers Controller (depicted in Figure 6.1) might look like the following.

```
http://offers.fedseeko.com/offers.xml
```

This URL tells the service to take Amazon and Evendi.com as vendors into account during the ranking and calculate the second results page for the query “ipod”. A returned offers list does not necessarily include offers from all available vendors since some calculated offer ranks might be too low to be included in the results set. A categorization component which is based on the Amazon Product Advertising API as well as the eBay Shopping API helps to detect the category for the ranking algorithm. Then, only the relevant online malls are queried either by using their Web Services or HTML-based web applications and corresponding results are returned in a unique format.

Each of the returned offers is identified by a clean REST URL and includes the basic details mentioned in Section 3.1 (product name, producer name, product picture URL, product description, offer price, detail page URL). Since only the first four details are product-specific, they are saved in a separate product model while the offer price and the detail page URL are saved in the offer structure pointing to the corresponding product as well as the selling vendor. Figure 6.3 shows a UML class diagram for the Offers Service.

![Figure 6.3.: Offers Service Classes.](image-url)
As can be seen, the system offers three different wrapper types. These include the 
\texttt{CssSelectorWrapper}, the \texttt{RegexWrapper}, and the \texttt{XPathWrapper}. Wrappers are used by the \texttt{Vendor} class to extract offers from a \texttt{ResultsPage}. According to Req 0.3 (section 4.1), results pages may be retrieved by external entities using vendor configurations that are made of a set of parameters. The ranking algorithms are implemented in the \texttt{Offer} class. They decide about which results pages are to be retrieved based on a detected category (Req 0.2) and a vendor’s suitability for that category (Req 0.1).

The Offers Service implements all techniques described in the vendor product information search section (section 4.1). It is made of about 5,500 lines of code. Taking into account that Ruby avoids unessential syntactic sugar and that the Offers Service is not the most important component of the system, its extent appears to be quite large. Anyhow, the implementation of extraction components supporting regular expressions, XPath, and CSS selectors as well as the comprehensive ranking component required more implementation work than expected. The Offers Service retrieves bootstrapping information concerning products of arbitrary categories. The Products Service further processes this information as to be presented below.

6.1.2. Products Service

The Products Service [172] is the central component of the federated product information architecture being presented in this chapter. It includes all algorithms of Section 4.2 and chapter 5. Like the Offers Service, it has been implemented based on Ruby on Rails with some Java extensions. Again, product information may be delivered as plain HTML, XML, or JSON. Figure 6.1 points out that the main components of the service are the producer site locator, the product specifications extractor, and the product specifications matcher. Additionally, a product crawler, the OWL importer, and the triple store are shown. All of these components are relevant for retrieving and managing technical product specifications. A query URL for the Products Service might look like the following.

\begin{verbatim}
http://products.fedseeko.com/products.xml
?product[name]=finepix%20j30&product[producer_id]=1
\end{verbatim}

As can be seen, the products list can be filtered by providing a name and a producer ID. The producer ID has to be detected in a previous step. Each product may also be identified by a clean REST URL. The service allows the creation of new products or producers through POST requests as well as updating and deleting products or producers. Everytime a product is created or updated, a retrieval job is added that uses the various internal classes to retrieve relevant technical specifications. In the following, the relevant components participating in such a retrieval job are presented. They include the product specifications page locator, the product specifications extractor, and the product specifications matcher. Since they implement the central algorithms of the previous chapters, each component is additionally described with a UML class diagram similar to the one of the Offers Service.
Product Specifications Page Locator

The most important classes for the producer site locating step are shown in Figure 6.4.

As can be seen, the Locator is the central component here. It offers methods for reranking SearchResults, locating product pages, and finding related product web pages. The page retrieval is based on a MetaSearch engine that uses several WebSearch engines to find potential product pages. Two such search engines are GoogleSearch and BingSearch. The meta-search engine implements the Borda ranking while the rest of the ranking algorithms are placed in the locator. Another important class is the WebCrawler that offers different methods to find concrete product specification pages for given product pages on the same domain. The WebPage class enables comparisons of web pages to find out how similar two potentially matching pages are. The URI class is one of many helper classes. One of its more important methods is the distance method that calculates a similarity value for two given URIs.

Req 1.1 (see section 1.4.3) is fulfilled since the locate_product method only needs a producer and a product as input. Req 1.2 is fulfilled as well since domain knowledge in the form of given product specifications is not required for the locating process. Finally, also Req 1.3 has been considered. The rank_search_results method provides only the best-ranked product specifications page.
Product Specifications Extractor

Having identified potential product specification pages on producer domains, the extraction component is activated for collecting concrete product specifications out of these pages. Its most important classes are shown in Figure 6.5.

The central class is the Extractor that initiates the process for a Product and a potentially given example specification contained in the product specifications list, hence, satisfying Req 2.1 (section 1.4.3). First, all Wrappers for the given product as well as its Producer are tried out. If no wrappers are available or no wrapper can be verified, the WebServiceAdapter is used to retrieve information on a web page’s elements by using the Web Pages Service. A short description of this service is given in the next section. Then, all elements are inserted into the Elements structure for being able to navigate in the element hierarchy.

The ClusterFactory creates the elements lists of type Cluster by taking given cluster options into account. Created lists are purged and some of them are dropped. The
clustering process is repeated two times to create groups of elements lists and finally retain the candidates for wrapper generation. If an example specification had been provided initially, it is taken into account during the clustering steps. Likewise, available domain knowledge given through the KeyPhrase class may affect the process of cluster selection. The selection also works without key phrases (Req 2.2). Having the most promising group at hand, the extractor first asks its best-rated list to generate an XPath wrapper configuration as described in the extraction chapter. If this process fails, the containing group is instructed to do so. In any case, the generated XPath queries are used to add a new Wrapper for the product of interest. Now the extractor can apply this wrapper and save the retrieved set of ProducerSpecifications to the database. A producer specification is a product specification in the producer’s terminology as claimed by Req 2.3. For every confirmed extraction process, the keys of the found product specifications are saved as KeyPhrases to be reused in the future.

Web Pages Service

The Web Pages Service [174] is an auxiliary service for retrieving information about web page elements. It offers an endpoint for querying URLs to be rendered in a browser and returns the information on contained HTML elements in XML. This service is only used by the extraction routine of the Products Service.

The Web Pages Service is implemented in Ruby on Rails. It is based on the FireWatir library, the Firefox browser, and a JSSH addon. In combination with this addon, FireWatir allows the remote control of Firefox as well as the execution of Javascript functionality against the page being currently rendered in Firefox. Thus, when being provided with a URL, the service starts a Firefox instance, navigates to the page being identified by the given URL, potentially clicks on a link for showing product specifications, and executes the Javascript functions for calculating all element positions and visibility settings. The information on these elements is returned in an XML document.

Product Specifications Matcher

The extracted product specifications are of various formats and need to be integrated based on the matching functionality described in the previous chapter. Therefore, a set of classes has been implemented, the most important of which can be seen in Figure 6.6.

The entry point is the MatchProcessor. In a first step, it uses the Categorizer for assigning a corresponding category to the Product. Then, the hierarchic matching approach can be started. Different elementary Matchers are executed to create numerous candidate ProductSpecifications based on the extracted ProducerSpecifications and modeled Properties as stated by Req 3.1 and Req 3.2 (section 1.4.3). These candidates include calculated similarities and thus represent the similarity matrices. Based on the candidates, the EvolutionaryMatcher and the NaiveBayesMatcher assess elaborate matching results. With their help, the AssociationOptimizer creates a set of 1-to-1-matches (Req 3.3) taking the stable marriages problem from section 5.3.5 into account. The remaining product specifications are marked as valid matches while all
other candidate specifications are marked as non-matches. Finally, the valid product specifications are cleaned by the Normalizer. The outcoming product specifications are comparable in the sense that they all follow the schema defined in the central ontology.

**Additional Products Service Extensions**

Before the ontology, which has been developed in section 5.1, is employed for product specifications matching, it has to be imported and mapped to a database schema. The domain ontology is used for describing the schema while the application ontology is already part of the database’s content. The approach of mapping the ontology to the database has been chosen for two reasons. Firstly, the major advantages of semantic repositories such as Sesame or OntoBroker over databases include the possibility of executing reasoning algorithms on the ontology as well as fast adaptations of the corresponding data schema. For the developed application, however, reasoning mechanisms do not provide any benefits and schema adaptations are unlikely since $o_d$ is not intended to be enhanced. Secondly, the Ruby on Rails framework allows simple database transactions through ActiveRecord [85] virtually avoiding the manual creation of SQL queries. This way, the code can be kept very clean and abstract. The adoption of a semantic repository would require SPARQL or F-Logic queries which prohibit a programmer from using standard persistence frameworks. Anyhow, the knowledge being imported into the database can easily be exported to OWL.
Evaluation

again. Thus, the reusability of the product information model and the included product information does not suffer from the chosen approach.

The Products Service includes a **ProductCrawler** as well. This crawler initiates the locating of a product page, followed by the extraction and the matching step. The product crawler is thus working with all classes of Figures 6.4, 6.5, and 6.6. Further components of the Products Service include a visual ontology editor as well as an ontology learning environment. The visual editor is shown in Figure 6.7.

![Products Service](source_url)

**Figure 6.7.:** Screenshot of the Products Service.

It is based on a JavaScript library called InfoVis [16]. When being initiated, it presents the main classes of the domain ontology to the user, that is, products, units, and languages. Starting from the product types, a user can navigate to sub-types of products and corresponding properties. She can add additional product types, properties, languages, etc., update existing concepts, or delete unused elements. This way, the ontology may be managed manually. The ontology learning component enables the comfortable adding of new product properties or property synonyms by using clustering algorithms being based on a vector space model. Both, the ontology editing and the
ontology learning component are byproducts of this prototype and have thus not been described intensively in the concept chapters.

Since the Products Service is the most important component of the whole system, it has also grown to be the most comprehensive one with about 12,500 lines of code. In the following, the Snippets Service will be presented briefly which provides information from third-party sources.

6.1.3. Snippets Service

The Snippets Service [173] is able to access dynamic web pages, query them for an arbitrary string, and extract corresponding results from a retrieved results page. Examples for such dynamic web pages are Google and TextRunner. Before accessing these sources, they have to be configured using CSS selector wrappers. The approach is quite similar to the one taken in the Offers Service. Anyhow, the configuration has to be done completely manually. Additionally, no ranking functionality is included. The results of queried dynamic sites are arranged in sets of key-value pairs. Thus, the application using the Snippets Service has to take care of how to present the provided information.

The Snippets Service is an attachment of the federated product information architecture and does not implement any relevant algorithms. Its size is about 2,500 lines of code. With a set of three product information Web Services at hand, applications are required that present the product information to a user in an adequate way. A corresponding web portal is described in the following.

6.1.4. Fedseeko

The last one of the theses provided in the introductory chapter states that integrated sets of product specifications allow effective product comparisons. This thesis correlates to the final step of the FEAD Chain, namely, displaying product information in an appropriate manner (Figure 6.8).

Figure 6.8.: The FEAD Chain - Present Product Information.
For being able to prove this thesis, a web portal going by the name of Fedseeko [169] has been developed that interacts with all previously described Web Services and presents offered information in an appealing manner. The interface is divided into two perspectives. In the first one, offers from different vendors are provided based on the Offers Service’ ranking values (Figure 6.9).

Several tabs allow searching on a single vendor’s database while the “All” tab takes all chosen vendors into account and asks the Offers Service to calculate the set of relevant products. As can be seen in the screenshot, a query for “i7” has been categorized to belong to the CPU domain. Search results from Amazon and Evendi.com have been ranked high since these shops are considered to be experts for this category. By clicking on the cross behind the vendor names, a user could remove vendors from her portfolio or add new vendors by clicking on the plus sign. The configurations are saved in the user’s account. If no user is logged in, this information is saved in a browser session.

When clicking on one of the retrieved offers, a detail view is opened presenting the corresponding product. For this view, the active tabs include one tab for the offer’s vendor, one tab for the product’s producer, and several tabs for the selected set of
third-party information providers. The producer tab is used to list product specifications being retrieved by the Products Service while the third-party tabs present information delivered by the Snippets Service.

Information being provided by the Products Service is cached in Fedseeko, thus allowing the effective comparison of products based on this information (Figure 6.10).

![Fedseeko's Product Comparison Interface](image)

**Figure 6.10:** Screenshot of Fedseeko’s Product Comparison Interface.

The product comparison is based on the product specifications themselves. The user may navigate to interesting products by iteratively clicking on clustered value groups being located in the facets field. The facets cannot be seen in the screenshot since they are automatically closed when only one, two, or three products are left. A breadcrumb navigation at the top of the page allows the user to view and edit selected facets. In the comparison view, all technical specifications being available for the remaining products’ category are listed next to each other. In this manner, it gets easy for the user to compare products based on their characteristics.

Since Fedseeko is the reference implementation for federating product information of all previously described services, it consists of about 11,000 lines of code. It is extended by the cross-site request plugin presented in the next section.
6.1.5. Fedseeko Browser Plugin

As already described in the concept, retrieving web pages from an online mall at a large scale while always using the same IP address is risky. This section describes the problem once again in detail and offers a practical solution. The packet flow for the scenario is pictured in Figure 6.11.

![Figure 6.11.: Provocation of Denial-of-Service Attack Reactions by the Offers Service.](image)

No attacker takes part in the communication with the online mall. However, the server in the middle representing the Offers Service might look like an attacker to the vendor application. Different solutions are imaginable to solve the described problem. The simplest idea would be to distribute the Offers Service, e.g., by deploying it in a cloud architecture that allows the requests to be executed from different network interfaces, each communicating through its own IP address. Although being a quite robust solution, the problem is only solved until a certain amount of users is reached, thus requiring more IP addresses to be allocated for the service. A second solution would be to provide the used IP addresses to the queried vendors for them to disable DoS attack reactions on these addresses. Such a solution is not feasible as well since the Offers Service should be easily extendable in a way that every user is able to add new vendor configurations which can be used by other consumers hereafter. The process would be massively limited if only contracted vendors were available. Additionally, connected vendors could not recognize simple DoS attacks anymore if they were executed through the Offers Service. Due to these reasons, a more complex solution had to be developed.

The idea is to not let the Offers Service itself request the results pages. Instead they should be retrieved by the web service consumer being provided with the different vendor configurations. Concerning the presented architecture, the consumer would be Fedseeko. Hence, the configurations need to be delegated one step further to the client’s browser. For being able to remotely query web pages through a browser on behalf of the accessed web application, the browser has to be extended with a plugin [175] allowing controlled cross-site requests. Figure 6.12 shows the sequence diagram for such a plugin.

As can be seen in the figure, the user’s browser first requests an intermediate loading page by providing a user-given query, vendors of interest, and a results page number (step 1). The request is forwarded to the Offers Service (step 2) which calculates a ranked list of relevant offers for the given parameters (step 3) and returns them to Fedseeko (step 4). Some of the offers may be empty, thus, Fedseeko detects the vendors providing content for these offers (step 5) and retrieves required vendor configurations from the
Offers Service (steps 6 and 7). Then, the loading page can be returned to the client’s browser, together with a set of web page retrieval instructions and corresponding vendor configurations (step 8). The browser parses these instructions and instantiates the plugin (step 9). Depending on the vendor configurations, the plugin executes a set of GET and POST requests on required online malls (step 10) and collects the returned vendor offers pages (steps 11 and 12). By using AJAX, the browser requests the offers list for the currently shown loading page and provides all downloaded web pages to Fedseeko (step 14). Fedseeko posts these web pages into the Offers Service’s database (step 15). Hereafter, it queries the Offers Service for the same offers list as before (step 16) which is now filled with actual data (steps 17 and 18). Fedseeko generates a corresponding HTML snippet for representing the offers (step 19) which are finally inserted into the loading page by the client’s browser.

The presented approach requires complex interaction mechanisms. However, as described above, it constitutes the only feasible solution for retrieving offers from vendors without provoking reactions to assumed DoS attacks. The plugin has been implemented in C++ based on the Qt framework [131]. It is available for all operating systems and major browsers including Mozilla Firefox, Microsoft Internet Explorer, Apple Safari, Google Chrome, and Opera. It accepts requests for executing GET requests based on URLs as well as doing POST requests based on a URL and a set of parameters. Cross-site requests are often used for attacking a user’s system by introducing bad code (Cross-Site Scripting). Therefore, the plugin allows such requests only for configured pages. Additionally, the user can set up the plugin to ask for permission each time a cross-site request is to be executed.

Since the plugin has intentionally been kept as generic as possible, it might also be helpful to other web applications. Mobile applications for cell phones do not need such
plugins as they can directly be designed to communicate with the product information services in an adequate way. One such example is presented in the next section.

### 6.1.6. Fedseeko Mobile

With the introduction of the first iPhone in 2007, Apple accomplished what other producers of mobile devices had not been able to put through: the day-to-day usage of mobile Internet by average consumers. To this day, competitors outgo themselves with new smartphone features perpetually, hence, proving the necessity of adapting available web applications to such mobile devices.

Especially caused by the difficulty of not being able to allow direct vendor requests through the Offers Service, a native iOS application [149] has been developed that consumes product information from the previously presented Offers Service and Products Service. A set of screenshots of the iPhone/iPod version are provided in Figure 6.13.

Figure 6.13.: Screenshots of Fedseeko Mobile.

The application flow consists of several steps. Initially, a splash screen is shown with the Fedseeko logo (first screenshot) and the currently available set of vendors is retrieved from the Offers Service. Then, a selection page appears where the vendors can be chosen and a query may be entered (not shown in the figure). Having submitted the query, the application provides some feedback on which steps are currently taken to retrieve potential offers (second screenshot) from the Offers Service. This also includes the interaction with chosen online malls and the handover of required results pages to the Offers Service. When the query hits are returned by the Offers Service, a corresponding list is displayed to the user (third screenshot). Furthermore, single products may be examined as can be seen in the fourth screenshot.

When inspecting a product in detail, the application automatically requests the Products Service to return technical product specifications that may be displayed in the
specifications tab. If no such information is available, the Products Service is prompted to retrieve it. A short section on lessons learned is given in the following.

6.1.7. Lessons Learned

The previous sections briefly presented the components deploying all described algorithms in a federated product information search architecture. During their implementation, several adaptations became necessary that could partly not have been foreseen since they were often related to peculiarities of the employed frameworks. Some of these lessons learned will be described in the following. The different aspects are divided into architectural, algorithmical, and framework-related findings.

Architecture

The architecture of the federated product information system has been laid out as a service-based one. This is mainly due to the separation of concerns as well as making collected product information reusable through a standard interface. The service paradigm of choice is REST for several reasons.

SOAP is clearly message-oriented while REST focuses on resources rather than function calls. The interactions with all services in the Fedseeko system include querying for entities (e.g., offers or products) using GET, creating new entities (POST), updating entities (PUT), and removing outdated entities (DELETE). PUT and DELETE are not implemented in some web application servers. A workaround thus consists of using the POST method with an additional parameter, e.g., _put. Further reasons for using REST are that SOAP contents are hardly readable for humans, REST is better suited for short-running services, and caching data is difficult in SOAP.

A significant finding consisted in the fact that service-oriented architectures produce a reasonable overhead in implementation and information handling. Naturally, an increased complexity for the system had been expected. However, the constant transformation of information from programming language-dependent data structures into reusable representations like XML or JSON and the following retransformation required the implementation of a standardized REST consumer being reused for all services. Originally, the Products Service had been divided into an extraction and a matching service which also communicated with each other in a REST-based manner. For complexity reasons, both components have been reunited in one component in the final prototype.

Algorithms

Some algorithmic peculiarities have only been found out during their implementation. For example, concerning the usage of clustering features in the extraction framework, regular expressions should be used with care since they may slow down the whole extraction process drastically. While the application of regular expression-based indexless XPath queries for group creation is reasonable (see section 6.22), the clustering of web page elements based on such expressions may double the time for wrapper induction.
With the help of a very product-specific configuration during the clustering-process of
the extraction step it was not possible to support some kinds of product specification
representations available on the Web. Adding additional grubby code could have solved
the problem for all found examples. However, the extraction algorithm has been kept
generic intentionally. It is possible to configure the clustering step in an arbitrary manner,
thus also supporting extraction tasks like finding all user posts in a board. For the sake
of reusability, confusing producer templates are therefore not supported. The overall
precision and recall values are still remarkable as the evaluation section proves.

Furthermore, the complete process of retrieving, extracting, and matching product
specifications had to be swapped into its own thread since it just takes too long for a
user to wait on the process completion. This has been achieved by a job worker based
on DelayedJob [47]. Each time, a product is queried which is not yet enriched with its
product specifications, a new job is created which is put into the job worker’s queue.

Frameworks

The major part of the different Web Services and web applications has been implemented
in JRuby on Rails and Java. The combination of both is quite powerful since Ruby allows
fast and clean development of functionalities while Rails enforces the development of
strictly MVC-based web applications. By sticking to the JRuby interpreter, not only
Ruby libraries (gems) may be included, but also all kinds of Java libraries can be reused
directly inside the executed Ruby code. This way, the best of both worlds, namely,
the flexibility of Ruby and the multitude of mature tested Java libraries, is combined.
However, since the first implementation work in 2008, Rails has been undergone a series
of drastic changes. Version 2.1 had been published in June 2008, followed by version
2.2 (November 2008), 2.3 (March 2009), and finally 3.0 (August 2010). In between,
countless patches have been published. Especially when switching between minor or
major releases, every developed application had to be refactored to work with recent
changes which caused a significant overhead. Since version 3, the framework may finally
be called technically mature. As Rails does also enjoy a broad user base, the original
choice can be justified.

A similar development has been undergone with JRuby. Extensive revisions have
been made from JRuby 1.1 (April 2008) to 1.6 (March 2011). Each JRuby update
required reinstalling and testing the employed libraries for compatibility. A considerable
improvement marked the introduction of RVM [155] (Ruby Version Manager) in late
2009. It allows installing different Ruby and JRuby versions with various gem sets in
parallel. Thus, web applications depending on new functionalities of the language or any
framework may be refactored while other applications can stay with their dependencies.

Concerning the libraries applied in the extraction framework, FireWatir [161] is to
be mentioned. As described above, it enables the remote control of the Mozilla Firefox
browser. FireWatir is based on Watir (Web Application Testing In Ruby). It has been
included in its own Ruby Web Service since it is not fully compatible with JRuby yet.
Direct alternatives for FireWatir are Watij/WebSpec [46] (Web Application Testing In
Java, a Java port of Watir), Selenium [96] (similar to Watir), and HTMLUnit/Celerity (a
GUI-less browser). FireWatir has been preferred over Watij and Selenium for its direct integration in Ruby. HTMLUnit and its Ruby enhancement Celerity had been originally chosen since they do not require the interaction with a real browser. However, it seems that the integrated browser functionality has not been fully developed yet as element coordinates of given web pages were not correctly calculated. Nokogiri [139] and Hpricot [184] are Ruby libraries for web page interaction. They do not allow the calculation of coordinates at all and could thus not be employed in the Products Service.

For the machine learning-based matching task, Knime [21] has been used to construct a corresponding workflow. The original idea of directly integrating the designed workflow as a Java library was not realizable. Thus, all functionality was reimplemented based on the Weka [54] library. Weka offers about 50 different machine learning algorithms. Since they all have the same interface, it has been quite easy to compare the approaches with each other. From 50 tested implementations, 22 produced results with reasonable $F_1$ scores and efficiency values. For these 22 learners, differences between $F_1$ scores and efficiency values have not been significant. Thus, the Naive Bayes matcher was chosen as a good representative. Available Ruby libraries for machine learning have not been employed for their lesser functional range.

Having outlined the system implementation, the next section finally presents evaluation results for the most important parts of the concept.

6.2. Evaluation

The service-based architecture presented in the last section allows the evaluation of all algorithms developed throughout chapter 4 and chapter 5. Therefore, this last major section makes use of the implementation for proving their feasibility. First of all, the most important evaluation measures are introduced briefly. Then, the gold standard creation process as well as the gold standard itself are described. As the theses provided in section 1.4.2 focused on locating producers’ product information pages, extracting technical specifications from these pages, and matching them with a central ontology, the subsequent evaluation only accounts for these steps. The processing of vendor information including presented ranking algorithms is not examined.

All evaluation algorithms have been taken out on a 32 bit Microsoft Windows 7 testbed running on an Intel Core 2 Duo CPU at 2.54 GHz and 3.5 GB of RAM as well as a 32 bit Ubuntu 10.10 Maverick Meerkat virtual machine with comparable specifications.

6.2.1. Evaluation Measures

The developed algorithms are located in the area of information retrieval and information extraction. Evaluating the effectiveness of such algorithms is usually based on a so-called gold standard as well as two measures being called precision and recall. Originally, a gold standard is a manually assembled test collection of documents, each with a binary classification as to be either relevant or non-relevant. In a broader sense, a gold standard contains the set of elements an algorithm should retrieve when working as intended by the developer. The mentioned measures, precision and recall, both are determined
by calculating the set of true positives (TP), false positives (FP), and false negatives (FN) concerning the gold standard. That is, elements being classified as relevant by the gold standard and retrieved by the algorithm are classified as true positives. The set of elements being retrieved by the algorithm although not being marked as relevant is called false positives. Relevant elements not being retrieved by the algorithm constitute the set of false negatives. Figure 6.14 visualizes the interconnection of the different sets.

![Interconnection of sets](image)

Figure 6.14.: False Negatives, True Positives, and False Positives.

For the sake of completeness, true negatives are to be mentioned. They contain all nonrelevant elements which are not retrieved by the examined algorithm. Now, precision and recall can be calculated. The precision $P$ estimates the amount of relevant documents in the set of retrieved documents (Formula 2.1).

$$P = \frac{|TP|}{|TP| + |FP|}$$  \hspace{1cm} (2.1)

As can be seen, the amount of true positives is divided by the total amount of retrieved documents, that is, the sum of true positives and false positives. The second measure is the recall $R$ being presented in Formula 2.2.

$$R = \frac{|TP|}{|TP| + |FN|}$$  \hspace{1cm} (2.2)

The recall calculates the percentage of retrieved relevant elements by dividing the amount of true positives by the total amount of relevant documents ($TP + FN$). Precision and recall are of equal importance. However, in general, optimizing a system for one of those measures is at the expense of the other one. The idea is to calculate a weighted harmonic mean $F_\beta$ that takes both measures into account while still allowing to focus on one of both. For this work, only the balanced $F_1$ score will be important. The resulting measure is shown in Formula 2.3.

$$F_1 = \frac{2PR}{P + R}$$  \hspace{1cm} (2.3)

The necessary gold standard is examined in the next section.
6.2.2. Gold Standard

All algorithms relevant for the current section are implemented in the Products Service and need to be evaluated based on a gold standard. Since the type of information collected through the Products Service is quite specific, no existing gold standards have been identified that would have allowed such an evaluation. Therefore, the Gold Standard Manager [170] has been developed which is to be described briefly in this section. A screenshot of the manager’s user interface is provided in Figure 6.15.

![Gold Standard Manager Screenshot](image)

Figure 6.15.: Screenshot of the Gold Standard Manager.

The Gold Standard Manager focuses on collecting information on product specification pages, extracted raw product specifications, and matched normalized product specifications. An initial ontology was imported into the manager and iteratively extended by different independently working people.
The collection process consists of randomly choosing a technical product (e.g., by clicking through the products from a technical category of a vendor like Amazon) including its name and its producer’s name, finding the URL of the correct product specifications page for this product on its producer’s domain, and deciding about the category of the product. If producer and category are already known, the product can directly be inserted into the manager. Otherwise, producer and category (and potentially also a parent category) need to be created first. Each time a product is created or updated, the Gold Standard Manager caches the assigned product specifications page including all images and related documents in the background, therefore allowing a complete offline evaluation of the extraction and matching components. The problem of changing page layouts and outdated products is also bypassed this way.

When a product is created, its raw specifications can be extracted from the provided page. Corresponding key and value fields allow inserting the specification strings exactly in the format used on the producer site. Then, the producer specifications can be matched with the properties available through the ontology. If there are not enough properties (and corresponding units) available, they have to be created first. The matching is to be executed through the interface shown in the screenshot (Figure 6.15). When all extracted specifications are matched, the process is finished.

The final gold standard collected through the manager is of considerable size. An overview is given in Table 6.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categories</td>
<td>37</td>
<td>Domain Instances of Concept <em>Product</em></td>
<td>Photography &gt; Digital Camera</td>
</tr>
<tr>
<td>Producers</td>
<td>48</td>
<td>Corresponding Producer of a Product</td>
<td>Fujifilm</td>
</tr>
<tr>
<td>Products</td>
<td>304</td>
<td>Application Instances of Concept <em>Product</em></td>
<td>Finepix J27</td>
</tr>
<tr>
<td>Raw Specs</td>
<td>9586</td>
<td>Product Specifications from Producer’s Web Page</td>
<td>Digital Zoom: Approx. 5.7x</td>
</tr>
<tr>
<td>Properties</td>
<td>984</td>
<td>Domain Instances of Concept <em>Property</em></td>
<td>Zoom &gt; Digital Zoom</td>
</tr>
<tr>
<td>Matched Specs</td>
<td>4935</td>
<td>Application Instances of Concept <em>Property</em></td>
<td>Digital Zoom: 5.7</td>
</tr>
<tr>
<td>Units</td>
<td>80</td>
<td>Domain Instances of Concept <em>Unit</em></td>
<td>times</td>
</tr>
<tr>
<td>Languages</td>
<td>3</td>
<td>Domain Instances of Concept <em>Language</em></td>
<td>English</td>
</tr>
</tbody>
</table>
The table consists of four columns. The first one displays the type of information being included in the gold standard. The second column lists the number of entities available up to this date. In the description column, the connection to the original ontology concept is explained if available. This is necessary since some renamings take place during the import process of the ontology. For example, categories are labeled as to be domain instances of the concept Product, that is, a category is an instance of a domain ontology concept and is therefore located in the application ontology still residing in the TBox. A product instantiates a category and is thus an application instance belonging to the ABox of the knowledge model. The last column provides an example for each type. The notation “Photography > Digital Camera” implies that DigitalCamera is a child category of Photography.

As the table shows, over 300 products have been collected with an overall set of about 10,000 extracted product specifications. About 5,000 specifications have been matched. A major set of product specifications provided by producers is just senseless, e.g., “Dial-A-Cycle: Yes” for a laundry dryer, and has been ignored during the matching process. About 1,000 properties sufficed to create the matched product specifications. It has been taken care to distribute the set of collected products over the total set of categories. Additionally, the amount of products per producer has been limited by two for avoiding inappropriate $F_1$ score ameliorations of the extraction routine caused by identical page templates. The distribution of products over given categories is shown in Figure 6.16.

![Figure 6.16.: Distribution of Products over Categories.](image)

Due to the preconditions of the collection process, the resulting gold standard is neutral concerning the peculiarities of developed algorithms and big enough to draw meaningful conclusions. Each algorithm is to be evaluated based on this data in the following sections.
6.2.3. Product Document Retrieval

The algorithms of the product document retrieval section will be evaluated first. The set of relevant documents consists of the product page URLs being available through the Gold Standard Manager. The set of non-relevant documents consists of all web documents except the relevant ones. The evaluation is therefore based on about 300 elements. In the following, the \( F_1 \) score will be calculated for different categories in the retrieval effectiveness section. Precision and recall will not be determined separately since, for this special evaluation task, they always reside on the same level. The effectiveness may vary for different categories, because employed search engines might rank product specification pages higher for products of categories with a wider consumer interest in such products’ specifications. The overall evaluation will nevertheless always remain in the domain of technical products. An analysis on the impact of available domain knowledge will be carried out as well since such knowledge should ameliorate the overall retrieval process.

In addition to the effectiveness, the efficiency is to be evaluated. This makes sense since additional crawling tasks during the product page identification might seriously slow down the overall process.

Retrieval Effectiveness

As mentioned before, the first evaluation has been executed without any domain knowledge consisting of product specification keys from previous extractions. The web page locating was therefore purely based on the heuristics presented in the extraction chapter. In Figure 6.17, the calculated measures for some selected categories are displayed. The categories have been chosen by their size. Categories with less than ten representatives in the gold standard are not displayed in the diagram.

![Figure 6.17.: Product Page Locating Effectiveness without Domain Knowledge.](image-url)
With a value of 0.67, hard drives exhibit the lowest $F_1$ score. This is mainly caused by the fact that for hard drives quite sparse information concerning technical specifications is offered. Consumers are generally only interested in the size of the hard drive while additional information is not too important. Corresponding web pages might therefore not be referenced that often and receive quite bad ranking values by public search engines. The best category concerning $F_1$ scores is the one of scanners. Again, this is reasonable since technical specifications are much more important for this category. Taking all 304 products into account, an overall $F_1$ score of 0.79 has been achieved.

Improvements would nevertheless be advantageous. Therefore, in a second cycle, a reasonable amount of domain knowledge has been added to boost the retrieval process. In the case of product page locating, domain knowledge consists of product specification keys from previous extractions that might be included in a potential specifications page and thus give a strong hint on which web page to choose. Figure 6.18 shows the results.

![Figure 6.18.: Product Page Locating Effectiveness with Domain Knowledge.](image)

It can be seen that the inclusion of domain knowledge improved the retrieval process. Especially, web pages for digital cameras, graphics cards, motherboards, and refrigerators have been retrieved at a higher $F_1$ score. Such products include quite specific key phrases which provide strong hints to potential candidate pages. The overall $F_1$ score could be improved by about six percentage points and reached a value of 0.85. With the help of more domain knowledge, this value might be further increased.

Extended white and black lists for URL scoring would also have helped in the retrieval process since many false positives included terms such as “news” and “support” in the respective URLs. Moreover, some errors were caused by outdated search engine indices, thus returning product page URLs not being valid anymore. Last but not least, unclean routing mechanisms on producer sites caused some errors. For example, some sites save navigation information in browser cookies. If following a direct product link from a search engine’s index, the respective page expects to be able to set a corresponding cookie in
Evaluation

the client’s browser. If this is not possible, for example, in the case of a programmatic web page access with limited cookie functionality, a redirect to the home page occurs that offers general information about a product’s company. Removing such abstruse behaviour on the producer’s side would enable product page retrieval for these sites, too.

However, for the principal part of the test set’s products, the locating component was successful. It has therefore been proven that product specifications pages can effectively be located in an automatic manner.

Retrieval Efficiency

Locating product specifications pages may demand a reasonable amount of time. Depending on the professionalism of the provider, querying web search engines is usually quite fast. In contrast, the extension of candidate sets by crawling related pages as well as downloading several pages for examining their contents can be slowed down massively by inertly responding producer servers. The diagram in Figure 6.19 provides an overview of how much time the locating of a product page requires.

![Diagram showing product page locating efficiency](image)

Figure 6.19.: Product Page Locating Efficiency.

The major part of carried out retrieval processes needed less than one minute with an overall average of about 20 seconds. The locating process is thus not to be judged as very performant. Anyway, for the reasons mentioned before, it is hard to improve on this fact since external factors such as Internet connection speed or server response times determine the duration of web page retrieval. As the whole process of retrieving, extracting, and matching product specifications has been outsourced into its own job worker, the extra cost in terms of time is not too critical.

Conclusion

The preceding sections examined effectiveness and efficiency of the developed and implemented product specifications page retrieval algorithms. It was shown that the retrieved results are of high quality even though it might take some time to find them. This has been a major requirement as all subsequent processes depend on the page locating component. With a growing amount of reasonable domain knowledge, the locating
process might even produce better results. Hence, the first thesis from the introduction chapter, namely, the automatic locating of product specifications on public producer sites, has been proven. The second thesis is to be examined in the next section.

6.2.4. Product Specifications Extraction

This section evaluates the process of extracting product specifications from previously retrieved pages. This time, the evaluation set consists of all manually extracted product specifications representing relevant elements as well as bogus data being extracted from given product pages representing non-relevant elements. For each of the about 300 products and 10,000 product specifications, the correlation of true positives, false positives, and false negatives is calculated to obtain corresponding precision, recall, and $F_1$ score values. The evaluation is split for different categories again since product domains with a stronger focus on product specifications might allow better extraction results caused by cleaner, more professional site templates. The best found clustering configuration is presented, too.

Moreover, the efficiency is evaluated as the use of regular expressions is expected to have a major impact on the runtime of the clustering algorithms.

**Extraction Effectiveness**

For extracting product specifications at the best possible $F_1$ score, the learning component has been fed with a set of promising clustering properties. Then, regarding a training set consisting of about 30 product pages and respective technical specifications, the learning algorithm chose the best fitting configuration which is listed in the following.

```plaintext
[create_lists]
  type="list" include_text=true regex_left_or_maximum_left=true
  indexless_xpath=true visible=true
[purge_lists]
  split_on_alternating_index=true
[drop_lists]
  has_alphanumeric_text=true has_min_size=5 has_varying_text=true
  has_varying_top=true is_not_noise=["select", "option", "a"]
  visible=true
[create_groups]
  type="group" regex_shortened_indexless_xpath=true
  regex_range_avg_height=200 regex_range_avg_top=200
[purge_groups]
  keep_biggest_items=2 sort={:right => "asc", :top => "asc"}
[create_candidates]
  type="candidate"
[purge_candidates]
  keep_biggest_items=1
```
As the configuration shows, basic web page elements are differentiated by their content being empty or not, by their left or maximum left coordinate, and by their indexless XPath query. In a second step, created lists are split if they consist of elements with alternating indices. Then, lists are dropped if they do not have alphanumeric text (e.g., they only contain numbers), consist of less than five elements, do not have varying top y-coordinates, contain noise tags in their XPath query, or are invisible.

The creation of groups is based on a shortened XPath query that may be altered dynamically through regular expressions, a range of accepted height values, as well as a range of accepted top y-coordinates. Then, for each group, only the two biggest items are kept which are to be sorted by their right x-coordinate and then by their top y-coordinate.

The final step consists of creating candidates out of all groups and returning the biggest group as extraction candidate. Using this configuration, the effectiveness results shown in Figure 6.20 have been achieved.

![Figure 6.20.: Product Specifications Extraction Effectiveness.](image)

Product pages with less than five product specifications have been excluded since such sparse information has been judged to be useless. Furthermore, duplicate extracts as well as inverted key-value pairs have not been regarded as errors as they are easily filtered out by the matching step. Again, the evaluation figure only includes the most important categories. The overall values represent precision, recall, and $F_1$ score for all products of the gold standard. It can be seen that a recall of 1 was reached for some categories, that is, the complete set of technical specifications was extracted for all corresponding products. Other categories performed worse, mainly due to confusing web page templates. It cannot be reasoned that beamers and printers are generally represented through bad-formatted HTML pages. Choosing different producers for these categories might already ameliorate the evaluation results heavily.

A further finding is that the extraction process’ recall is never below its precision. This makes sense since, in most cases, the routine either extracts all correct product
specifications or none of them. The recall is therefore either 0 or on a quite high level while the precision of a successful extraction step might still be impaired by additionally extracted bogus data not belonging to the actual product specifications. This is a desired helper since a low precision value is balanced by the subsequent matching step filtering out all non-matching product specifications while low recall values could not be balanced this way.

The overall $F_1$ score reaches 0.817. Since the recall is the most important measure here, it is beneficial that it reaches a value of 0.843. As for the locating component, a satisfying extraction effectiveness has been achieved. The intensive web page processing may again be at the expense of the efficiency which is evaluated in the next section.

**Extraction Efficiency**

Like the web page retrieval, the clustering process for extracting product specifications based on different page representations may demand some time. Hence, an investigation of the procedure’s efficiency has been taken out. In the first study, the previously mentioned clustering configuration for group creation has been changed to use the property `shortened_indexless_xpath` instead of `regex_shortened_indexless_xpath`. The evaluation results displaying costs in terms of time for the extraction process are shown in Figure 6.21.

![Figure 6.21.: Product Specs. Extraction Efficiency without Regular Expressions.](image)

The overall extraction time has been limited to 300 seconds, thus, some extraction routines have not been able to complete. However, an average runtime of 94 seconds per extraction has been measured during evaluation. Furthermore, as the figure shows, the majority of the extraction processes finished in less than 100 seconds. The effectiveness was a little worse compared to the extraction results being based on the configuration of the previous section.

The efficiency had been expected to downgrade when using the `regex_shortened_indexless_xpath` attribute. Therefore, a corresponding evaluation has been taken out. Results are shown in Figure 6.22.

Interestingly, the efficiency has not been impaired at all. Quite the contrary, it even improved. The average extraction cost decreased to 58 seconds. Again, the major part of
the extraction routines needed less than 100 seconds. It stands out that several peaks of the given bar chart are located around the 15 seconds mark while such local maximums seem to be rather uniformly distributed over the first 100 seconds in Figure 6.21.

Generally, using regular expressions for comparing potential element IDs is more expensive than simple string comparisons. The explanation for the observed reality lies in the way the shortened indexless path property is handled for creating groups. The extraction routine executes the clustering process several times for shortened XPath expressions of different lengths to find optimal clusters. Since regular expression-based IDs are far more flexible and find better clusters in less iterations, the clustering needs to be executed less often and therefore finishes in less time.

When clustering basic web page elements, the use of regular expression-based XPath queries is not recommended. This is due to the larger size of lists in comparison to groups. Hence, the multitude of regular expression-based comparisons would outweigh the performance gain caused by fewer clustering iterations.

However, the use of regular expression-based XPath queries proved to be a feasible approach for clustering groups. The automatically learned clustering configuration presented in the previous section has therefore also been approved from the efficiency perspective.

**Conclusion**

As the evaluation of the extraction component proved, the algorithms show feasible results. The focus has been on the effectiveness again which allowed the coverage of many different producer templates. From all three page representation formats, the visual representation provided the most valuable hints on where to find product specifications on a web page. It is therefore indispensable for a state-of-the-art extraction algorithm to employ such information.

The quite high costs in terms of time are acceptable since the process is running as a background task and may even be parallelized for several products, e.g., through a customized MapReduce algorithm [45]. Anyway, the second thesis in section 1.4.2 stating that high-quality product specifications can be extracted automatically has been proven.
Thesis three is in the focus of the next section.

6.2.5. Product Specifications Matching

For the integration of product specifications with the central ontology, a set of element-level matchers as well as two composite matchers have been developed. It is the goal of this section to compare all element-level matchers with each other as well as to evaluate the evolutionary matcher and the Naïve Bayes matcher.

The evaluation is taken out in several steps. First of all, the set of about 5,000 matched product specifications from the gold standard is used as the relevant elements. Then, another 5,000 false matches are randomly created as non-relevant elements. Iteratively, various evaluation sets are created with varying sizes of the training and test set. The first training set consists of 5% of the overall element set, thus leaving 95% for the test set. The training set percentage is increased by 5% for each iteration. This way, the impact of a growing training set on the overall effectiveness of composite similarities can be estimated.

Furthermore, an efficiency comparison of the composite matchers is carried out. The following sections describe the results of the evaluations.

Matching Effectiveness

The effectiveness is again based on precision, recall, and $F_1$ score. First, the elementary similarity measures are rated. Hereafter, the composite similarity measures are examined.

Elementary Matching Effectiveness The elementary effectiveness has been evaluated for all 11 similarity measures introduced in the previous chapter. Since a development of the similarities over a growing training set ratio would have produced quite confusing diagrams, the evaluation is limited to a typical training set size of 70% and a test set size of 30% of the evaluation data. Moreover, the thresholds calculated by the evolutionary learning algorithm have already been used for this evaluation. The results are shown in Figure 6.23.

As can be seen, the similarity measures achieve a quite high quality level. The precision of the identity similarity is naturally the highest since identical names of extracted product specifications and modeled properties virtually always imply the detection of a match. The recall of the identity similarity is the lowest of all name matcher similarities. This is comprehensible since one varying character in the given strings already sets the identity’s similarity to zero. The other five name matcher similarities have comparable effectiveness values. Their precision is lower than the identity similarity’s one since similar strings might be detected as to be identical by the Levenshtein distance or n-gram comparisons. Higher recalls are the consequence as near-duplicates can be found this way. With $F_1$ scores above 0.9, the word-level 1-to-2-gram similarity as well as the character-level 2-to-8-gram similarity offer the best results. The chosen character-gram lengths have been discovered to be optimal in a pre-evaluation and also represent typical n-gram sizes.
The structure similarity disappoints with a $F_1$ score marginally above 0.2. Also the data type similarity does not reach more than 0.495 $F_1$ score. These values could be ameliorated by extending the set of regular expressions for the detection of structures and data types. However, their effectiveness is limited by the fact that scalar structures and string data types are the most frequent ones. As mentioned in the information integration chapter, scalars and strings do not induce any similarity value since they cannot be detected by special patterns and do not have any explanatory power.

The remaining three similarities, namely, unit, unit domain, and keyword similarity, at least offer high precision values. All three similarities search for emerging strings representing units or typical keywords for a property’s value. The precision of the keyword similarity depends on the chosen minimum frequency of strings found in the training set values to be accepted as property keywords. All three similarity measures could easily be boosted by creating a broader training set. Furthermore, their recall will grow over time since more units and keywords are collected with executed matching tasks.

In summary, the quality of the developed elementary matchers is very satisfactory. As expected, the most significant similarities are the ones based on product specification keys and property names. It is therefore of importance to use all keys of extracted and matched product specifications as synonyms for the ontology’s properties. With a growing size of these synonym sets the similarities will gain even higher $F_1$ scores and thus allow the effective matching of product specifications and properties.

The basic similarities have been combined in two different composite matchers. Their results are shown in the following.
Evolutionary Matching Effectiveness  The evolutionary matcher has been evaluated by calculating precision, recall, and $F_1$ score for growing training set ratios. It combines all given similarities by taking a set of weights and thresholds into account. The thresholds have been determined by trying out different threshold ranges and iteratively refining the actual threshold to reach the optimal $F_1$ score for each similarity. The weights of the different similarities have been determined by assigning the precision value corresponding to the previously found $F_1$ score to each similarity. The resulting values might therefore vary for differing training set ratios. The corresponding diagram is shown in Figure 6.24. The evaluation measures have been set to zero for a training set ratio of 0% as the automatic configuration learning requires training data.

As can be seen, the precision reaches a value of more than 0.9 for a 5% training set size. This is intuitive as a small training set does not allow many false positives. The recall starts at a reasonable lower level of about 0.7. With the growing training set, the recall gets continuously better and finally reaches a value of over 0.94. The precision turns into a value of about 0.93 for a training set ratio of 95%. This is a little lower than the best precision value being reached for a training set ratio of 35%. The massive variation of the precision value for smaller training set ratios originates from new matches being added to the training set that might not yet be balanced with false matches of the same producer specification and property.

The diagram shows that the calculated configurations combine similarity measures such that at least the quality level of the best elementary similarity is reached. The calculated composite similarity does therefore not outperform similarities like the character-level 2-to-8-gram similarity explicitly. However, as mentioned before, a lower training set ratio impairs the name matcher similarities massively. The composite similarity would be automatically adapted in this case to base its calculations on other, less training set-dependent similarity measures. Therefore, the additional expenses for calculating
composite similarities are fully justified. Moreover, potential new similarity measures could directly be included in the overall similarity measure and upgrade the matching process without major revisions.

**Naïve Bayes Matching Effectiveness** Concerning the Naïve Bayes matcher, the effectiveness evolution over a growing training set has been evaluated as well. For each training set ratio, the learning algorithm detected the best similarities configuration of the Naïve Bayes matcher based on the optimal $F_1$ score. The configuration may vary for a growing ratio since some similarities, namely all name matcher similarities as well as the keyword similarity, heavily depend on a broad training set while similarities such as the structure and the data type similarity do not perform better when provided with more examples. The results of the evaluation are shown in Figure 6.25. Again, the evaluation measures have been set to zero for a training set ratio of 0%.

![Graph showing Naïve Bayes Matcher Effectiveness](image)

Figure 6.25.: Naïve Bayes Matcher Effectiveness.

As the graphs prove, the precision stays to a greater or lesser extent always on the same level. The recall grows with the training set ratio for the previously mentioned reasons. With a training set size of 70%, the optimal matcher configuration consists of the identity similarity, the word-level 1-to-2-gram similarity, the character-level 2-to-8-gram similarity, the data type similarity, and the unit similarity.

The $F_1$ score passes 0.9 for a training set ratio of 50%. With a score of 0.936 at 70% training set ratio, the overall Naïve Bayes effectiveness lies curitly above the best elementary similarity as well as the evolutionary matcher. Therefore, it offers the best effectiveness value of all matchers.

The different diagrams showed that both composite matchers represent feasible approaches while their results do not differ heavily. A further comparison concerning the efficiency is therefore conducted in the following.
**Matching Efficiency**

Both matchers have been found to perform considerably well. The Naïve Bayes matcher needs about 2.3 seconds for finding and rating a given configuration. Since the “finding” only consists of trying out a binary combination of similarities, this is comprehensible. The evolutionary matcher needs about 9.8 seconds for finding the optimal thresholds and weights for a binary configuration.

However, concerning the overall learning component, the evolutionary matcher outperforms the Naïve Bayes since it needs less than two minutes to find the optimal thresholds and weights for its total configuration on a training set of about 7,000 elements. The Naïve Bayes matcher generally needs more than 15 minutes for finding the best similarities configuration out of 11 given similarity measures on the same training set size. This is due to the amount of iterations needed for finding this configuration. The evolutionary matcher does 11 iterations, one for each given similarity, plus one iteration for the overall similarity. The Naïve Bayes matcher tries out all possible configurations of the available similarities. For a set of 11 similarities, this sums up to 2048 configurations. The difference between both composite matchers therefore lies in the learning complexity being linear for the evolutionary matcher \((n + 1)\) with \(n\) similarities) and exponential for the Naïve Bayes matcher \(2^n\) with \(n\) similarities). The number of adopted similarity measures for the Naïve Bayes matcher should thus be chosen carefully.

It is furthermore important to decide whether the combination of similarities should be adapted regularly or if one configuration might work for a longer time period. In the first case, the evolutionary matcher should be preferred for its fast configuration. In the second case, the Naïve Bayes matcher is superior since its \(F_1\) score is slightly better with a reasonable training set.

**Conclusion**

As the previous evaluation showed, the matching algorithms produce promising results. Each elementary matcher accounts for some hints on which product specification should be matched with which property. With no or very small training sets at hand, static similarities like the structure-based, data type-based, or unit-based one are of great help. Similarity measures being based on string comparisons are always of great help. However, they unroll their full potential only with training sets of considerable size.

Combining similarities produces even better \(F_1\) scores. The evolutionary matcher uses adapted weights and thresholds while the Naïve Bayes matcher simply decides about applying a similarity or not. Both matchers have their right to exist since given elementary similarities are optimally combined to gain even higher \(F_1\) scores. The Naïve Bayes matcher is a little better in terms of effectiveness while the possibility to configure the evolutionary matcher manually without having any training set at hand makes it the better choice for a system starting with the information integration from scratch. Additionally, the automatic configuration based on a training set is much faster for the evolutionary matcher since it only needs \(n + 1\) configuration runs with \(n\) being the number of elementary similarities.
Hence, the evolutionary matcher should be used at system initiation since the configuration is fast and can be adapted constantly with a growing set of user-confirmed data. As soon as the configuration changes per time unit fall below a certain threshold, the matching system can switch to the Naïve Bayes matcher which offers even better $F_1$ scores and builds on a widely accepted machine learning algorithm.

In any case, the evaluation proved the third of four theses presented in the introduction chapter. Product specifications can be harmonized using a comprehensive product ontology and thus allow effective product comparisons.

### 6.2.6. Comparison with Competitors

Fedseeko is not the first platform to gather product information from various sources. As mentioned in the introduction chapter, many other providers offer federated shopping portals (Ciao!, Yatego, Shopping.com, etc.). The information set being gathered per product, e.g., by Ciao!, is already quite impressive. It includes a collection of shops offering the respective product, related products, user opinions, a description, and sometimes even technical product specifications.

The most important competitor is Google Products. Product specification sets offered by Google are extremely neat. Their distinct homogeneity and remarkable completeness covering various languages suggest that major handcraft is applied by Google or the producers themselves. Naturally, such an approach outperforms the quality of Fedseeko’s product specifications. However, the disadvantage of manually created product specifications is that the immense number of available products does not allow a complete coverage. Especially less popular products from smaller companies tend to be left out. The algorithms presented in the previous chapters do not differentiate between popular and less popular products. Thus, a basic comparison of Google Products and Fedseeko has been established by counterchecking the availability of the randomly selected product set from Fedseeko’s gold standard in Google Products. A corresponding diagram is given in Figure 6.26. Again, only the most important categories are displayed while the overall value considers all products from the gold standard.

As can be seen, Google offers product specifications for less than 40% of the gold standard products. Fedseeko on the other side does not rely on manually collected information and may potentially present product specifications and additional information for every product being adequately represented on the Web. This is a major advantage.

However, the differences between existing federated product information platforms and Fedseeko do not allow to always prefer one platform over the other. If one relies on the correctness of product information, an existing platform might be the better choice. For having a wider set of products at one’s disposal with a steadily growing quality of included product information, Fedseeko would be the platform to use.

### 6.3. Conclusions

The current chapter described the implementation of all algorithms in a federated product information system as well as the evaluation of the most important algorithms with the
The implementation section first gave an overview of the federated product information system Fedseeko in general. Then, each component, namely, the Offers Service, the Web Pages Service, the Products Service, the Snippets Service, and finally the Fedseeko web application including a browser-based plugin as well as an iOS application for mobile product information search was presented. The Offers Service includes all algorithms for vendor product information search. Online shops may be configured through Web Services or web application wrappers. Queries are categorized, corresponding results are ranked and returned to the querying instance. The Web Pages Service is a component for remotely controlling a Firefox instance. It enables the visual analysis of web pages through their coordinates. The Products Service is the most important component since it implements all algorithms concerning producer product information search. Thus, a library for finding product pages on producer sites presenting corresponding technical specifications is included as well as an extended categorization module, an extraction routine, and the matcher architecture. All of those have been visualized with the help of UML diagrams. The Snippets Service is an additional component for including information from third parties. It is not in the focus of this chapter. Last but not least, Fedseeko’s interactive applications have been presented. The web application uses all of the available Web Services and allows effective product comparisons through a faceted search interface, hence, proving the fourth thesis from the introduction chapter. Based on the browser plugin, online shop requests can be executed through the client’s IP address. The iOS application has also been presented as an additional component for mobile interaction with the Web Services.

The evaluation section first introduced basic information retrieval measures as well as the Gold Standard Manager being implemented for collecting product information manually. Based on such information the evaluation could be accomplished. The main
focus of the evaluation was on the components being implemented in the Products Service. First of all, the locating algorithm has been evaluated. It produced feasible results since more than 85% of all gold standard products could be assigned with their corresponding product page including valuable product specifications. The usage of domain knowledge represented an important improvement. The evaluation of the extraction showed that its effectiveness is more than adequate for the task of product specifications retrieval from producer pages. An overall $F_1$ score of 0.817 has been reached while the recall even resided on a higher level. Thirdly, the matching component has been evaluated. $F_1$ scores over 90% for both composite matchers as well as good efficiency values for the evolutionary matcher proved the third thesis of the introduction chapter. It is therefore feasible to argue that extracted product specifications can be integrated with an adequate ontology. A final comparison with Google Products depicted the additional value of Fedseeko since existing federated product information platforms are still based on embracing manual efforts and hence do not cover the majority of products being presented on the Web.

The evaluation may thoroughly be called successful as all theses of the introduction chapter have been proven. The last chapter will provide some conclusions of the whole work and offer an outlook on future work.
Complemented by the final evaluation, the previous chapters developed a series of algorithms allowing the effective search of product information over various web sources as well as the integration of such information in a federated product information system architecture. The development has been grounded in a solid theoretical foundation given by the analysis of related work. A short summary of all chapters is provided in the following before the conclusions and future work finish up this final chapter.

7.1. Summary

The introduction chapter (chapter 1) focused on the way people use the World Wide Web for product information research. With the help of several surveys, it proved the importance of electronic commerce for the average consumer and presented the current manner of online product information research. It pointed out the necessity to improve the present research paradigm’s lack of structured assistance by automatically assembled product information. More precisely, one main case and four sub-theses were set up which pointed out that, especially by the use of information extraction techniques and semantic technologies, an all-embracing product view may be created.

Thus, the theoretical foundations for enabling a corresponding amelioration were established in the basics chapter (chapter 2) by focusing on federated information systems in general. The lowest layer, handling the information access, dealt with techniques of document retrieval, federated search, and federated ranking. Based on collected documents, the information extraction section presented techniques from the areas of structured, semi-structured, and unstructured IE. Since people search for product information on the Web, and HTML pages already offer some basic structure, the focus has been laid on IE from semi-structured sources. The information integration layer has been presented by the use of ontologies and ontology matching techniques. The ontology matching compromised a major part of this section since schema-based, instance-based, and combined matching techniques are useful for integrating product information. The basics chapter closed with some general information on product information representation.

How may the introduced techniques and technologies be combined and extended in order to refine the process of product information research? This question was answered on a high level in chapter 3. It presented the FEAD chain which offers a workflow for
gathering bootstrapping product information and iteratively extending and refining this information for the final presentation.

The information extraction chapter (chapter 4) eventually delved into the details of retrieving valuable product information from different sources. The first section focused on vendor product information search, that is, the gathering of basic offer information sets from online vendor platforms either by using Web Services or by directly extracting the information from appropriate web applications. Algorithms for ranking such information and legalizing the reuse of vendor information by swapping certain retrieval tasks to the client side have been presented as well. A more important section was that which constituted the producer product information search since it focused directly on the retrieval of web pages presenting product specifications on producer sites as well as extracting these specifications in a largely unsupervised manner. Here, the retrieval part mainly focused on the extension and reordering of search engine-based candidate lists while the extraction part introduced an unsupervised clustering-based algorithm with a learning component.

Product information may be represented in various formats and granularity levels. The product information integration chapter (chapter 5) therefore presented indispensable techniques for product information representation, categorization, matching, and normalization. On the one hand, product information representation was clearly the focus of this chapter since ontologies settled on different abstraction levels allow the reusable modeling of product information, e.g., by the use of OWL. The matching task, on the other hand, had been considered to be of utmost importance since extracted product specifications have to be integrated with the information model somehow. A combination of element-level similarities to calculate composite similarities was used to solve the problem.

A lot of conceptual work has been done in chapter 3 to 5. However, the feasibility of the different approaches can only be irrefutably proven by implementing and evaluating them based on scientifically recognized measurements. Therefore, the previous chapter (chapter 6) summed up the overall architecture of the prototype as a distributed web service-based application. Some lessons learned during the implementation process were added as well. The actual evaluations proved the feasibility of the most important algorithms with the help of effectiveness and performance measures. The $F_1$ scores achieved for each examined domain, namely, the producer page retrieval, the product specifications extraction, and the product specifications matching fulfilled all expectations and allowed the algorithms to be called effective.

So, what are the conclusions of this work? The next section will answer this question in detail.

### 7.2. Conclusions

A famous quotation that Albert Einstein is credited with states: “Know where to find the information and how to use it - that’s the secret of success”. Indeed, this can be seen as the maxim of the whole work laid out in the preceding chapters. Product details are
spread all over the Web and it is currently the task of a potential consumer to reunite relevant information for her product of interest.

As proved in the evaluation section, the initially described aspired manner of product information research is possible through the adoption of developed techniques. Fedseeko dynamically allocates and requests relevant product information sources and creates detail views with all-embracing product information sets for an arbitrary product of interest, especially focusing on the technical domain. The consumer is thus relieved from the task of manual information collection and integration. This is a major enhancement since the original complexity of the researching task was attended by additional expenses in terms of time taken up. Moreover, the expanded information base offered through a facetted search interface allowing semantic product comparisons enables optimized purchase decisions, since the ultimately chosen product perfectly fits the consumer’s desires.

The introduction chapter phrased two periods of the World Wide Web as Web 1.0 and Web 2.0. Such versioning terms may be seen as buzzwords since a heterogeneous structure like the WWW is not comparable to a standard software project. However, both terms have achieved a certain level of acceptance. Consequently, the next version going by the name of Web 3.0 is agreed to be based on semantic technologies enabling machines to understand the contents of web pages without human intervention. Ontologies and related concepts build the fundament of this new paradigm. Since Fedseeko is based on semantic product information representation, it is perfectly prepared for this new Semantic Web. This is also emphasized by the web service-based architecture since arbitrary applications may reuse the product information in combination with the OWL ontologies.

In summary, all set up theses have been proven. Certainly, the algorithms are not fully developed and the implementation is still prototypical. However, a major amelioration of the product information search paradigm has been achieved. Existing research work has been reused wherever applicable and extended with major innovations, especially in the information extraction area. Supplementary features would still be advantageous. Therefore, the final section provides an outlook on some outstanding tasks that have not yet been accomplished.

7.3. Future Work

The described set of product information search algorithms offers major enhancements for mitigating the problem of effective product information research. Various components could nevertheless be optimized or extended. For example, in the area of vendor product information search, the inclusion of a new vendor platform still requires some manual work. A user must locate a potential shop and label certain parts of the query page as well as the results page. In a fully developed system, the identification and analysis of new shop candidates could be fully automatized. For example, the structure of a results page could be detected automatically by using the clustering algorithm presented in the extraction section with an adequate configuration. In this way, changing page templates
would also no longer constitute a problem since the corresponding wrapper would be adjusted automatically. With the further development of Fedseeko, this functionality might be implemented.

Another point concerning vendor inclusion consists of emerging legal problems in reusing the page contents of these vendors. By outsourcing the page retrieval, the actual communication with a vendor page is executed by the platform user’s computer or mobile device. However, contents of online malls are still displayed on the Fedseeko web page which might be at odds with a vendor’s copyright.

Concerning the extraction of product specifications from producer pages, the set of available clustering properties could be further extended. The clustering algorithm has been kept as generic and extensible as possible to fit for multiple extraction scenarios. However, pertaining to the clustering properties, it is not yet complete. Analyzing the web pages the algorithm was not able to handle could form new ideas for additional properties.

The knowledge of Fedseeko and all its Web Services is already being shared with participants of the Semantic Web since both ontologies are published on the respective web pages. The next step would be a reasonable integration of these ontologies with eClassOWL and GoodRelations by Martin Hepp. This would especially encourage people to build applications based on these ontologies, thereby cross-linking Fedseeko with other entities on the Semantic Web.

Many surveys prove that third-party information such as user-generated content in the form of product opinions has a major impact on the consumer’s buying decision. Third-party information has only been considered marginally. The success of a federated product information platform depends on such information. Therefore, a first prototype for extracting product features and assessing attributes has been developed that allows the presentation of such content effectively. Future development should focus on this component.

Finally, the best way to evaluate a platform implementing all presented algorithms is by consulting its users. A serious assessment can only be done after having collected a reasonable amount of user data accumulated during its usage. Since Fedseeko is still in a prototypical state and its awareness level is very low, this kind of evaluation must be postponed.

In any case, the embracing set of developed algorithms for locating, extracting, categorizing, ranking, matching, and normalizing product information from sources being distributed all over the Web, by now offers a great simplification of the consumer’s product information collection task. Their development and implementation has therefore been worthwhile. Their publication through Fedseeko and continuing enhancement in the future puts gathered knowledge at everybody’s disposal and represents a little contribution on the way to the next level of the World Wide Web: the Semantic Web.
This appendix contains some additional information related to the developed algorithms. Firstly, information on how to interpret the used pseudo code is given. Then, all clustering properties of the product specifications extraction algorithm are listed.

A.1. Pseudo Code

The pseudo code used throughout the thesis is quite close to the Ruby programming language. In the following, the most important constructs are presented.

**variable name** A variable can be represented by any kind of string except the keywords presented here. A variable starts with a non-numeric character and is written lower case with underscores between the different parts of the variable name. Variables do not have to be declared.

**Object.method_name(Object)** A method follows the same name conventions as a variable. It is called on a class or an object and may have arguments surrounded by brackets. If a method belonging to some class is called from inside this class or an object instantiating this class, the name of the object on which it is executed may be left out. If a method does not have any arguments, the brackets can be left out as well. Methods execute some defined control sequence and may return a result. Method definitions are not needed in the pseudo code. Some important methods are presented later-on.

**Collection** Arrays and maps have some functionalities in common which are to be presented here. They both offer iterating through their elements using the “each” function that is followed by a so-called block defining what to do with each contained object. The block starts with “do” or “{” followed by the declaration of the variables given by “each” (e.g., |element| for an array or |key, value| for a map). Then, the actual instructions do something with the objects and the block is closed by “end” or “}”. Collections also offer the “map” function which works quite similarly, just that the results of the block are directly mapped to the corresponding objects. A collection may be extended with additional elements using the “<<(Object)” operator.
[Object,...] An array is an ordered collection of objects represented by comma-separated values in squared brackets. Arrays may contain different kinds of objects. Arrays offer methods such as “include?(Object)” to check whether an element is included in the array, “length” for the amount of included elements, or “[](Integer)” for retrieving the element at the position given by the Integer value. The empty array is represented by “[]”.

{Object => Object,...} A map is an ordered collection of key-value pairs where the key points to its corresponding value. Like arrays, maps can contain all kinds of objects. A map offers methods such as “keys” for retrieving the map’s keys as an array, or “[](Object)” for returning the value corresponding to the given object.

Object.=(Object) The equal sign is a method that assigns some object to the object in front of the sign.

Object.+=(Object) The plus-equal method combines the objects in front and behind both signs and saves the result in the first object.

Object.||=(Object) The or-equal method assigns the given object to the first object if the first object is null.

A.2. Extraction Algorithm Properties

The extraction routine for finding technical product specifications in corresponding product pages is based on a clustering algorithm. For creating the clusters’ IDs as well as purging and dropping clusters, a set of properties has been designed that may be determined through small code snippets. The property set is simply extendable by creating additional functions which are automatically made available to the overall clustering algorithm.

The current property set is listed below. It consists of clustering properties, purging properties, and dropping properties.

A.2.1. Clustering Properties

The clustering properties allow the creation of actual cluster IDs. They control the most significant part of the clustering process and therefore build the biggest group. Some properties are constrained to be applied on simple items while others require cluster items to disclose their potential. Since the major part works with both item types, no differentiation is effected.

<measure> <measure> is a place holder for the six basic measures being retrieved through the rendering component for each HTML element: left, top, right, bottom, width, and height. left represents the \( x_{\text{min}} \) coordinate, top the \( y_{\text{min}} \) coordinate, right the \( x_{\text{max}} \) coordinate, and bottom the \( y_{\text{max}} \) coordinate.

Example: width="20"
**maximum_<measure>** One of maximum_<measure>’s instances, namely maximum_width, returns the width of the current element’s highest ancestor not including additional text compared to the one being contained in the current element.
Example: maximum_width="56"

**avg_<measure>** An instance of avg_<measure>, e.g., avg_width calculates the average value of all included items’ width values.
Example: avg_width="57"

**regex_<measure>_or_maximum_<measure>** regex_width_or_maximum_width combines both included measures in a regular expression.
Example: width_or_maximum_width=".*(40|65).*"

**regex_range_<measure>** If instantiating regex_range_<measure> with regex_range_width, a regular expression describing a range around the item’s width would be returned.
Example: regex_range_width=".*(19|20|21).*"

**regex_range_avg_<measure>** A property like regex_range_avg_width returns a regular expression describing a range around the average width. The range size can be controlled by a parameter.
Example: regex_range_avg_width=".*(55|56|57|58|59).*"

**text** This property returns included text.
Example: text="Specifications:"

**inner_text** This property returns all text that may be retrieved for the current item.
For an HTML element, this is the element’s text as well as all text being included in child elements. For a cluster this is the sum of all items’ inner texts.
Example: inner_text="Specifications:Effective Pixels:10 million..."

**include_text** The include_text property returns true if the item contains text.
Example: include_text="true"

**visible** This property checks if the inspected element (or all item’s of the inspected cluster) is visible.
Example: visible="true"

**size** The size property returns the number of items included in the current item. If the item is an HTML element, its size is 0.
Example: size="24"

**inner_size** The inner_size of an item returns the sum of all inner sizes of items being included in the current item.
Example: inner_size="231"

**xpath** The xpath property simply returns an item’s XPath query. Clusters also have XPath queries being combinations of their items’ XPath queries.
Example: xpath="/html[1]/body[1]/div[1]/div[2]"
shortened_xpath  Controlled by a length parameter, this property returns a shortened XPath query. Sometimes, shortened XPath queries are better to describe cluster items.  
Example: shortened_xpath="/html[1]/body[1]/div[1]"

indexless_xpath The indexless_xpath returns the item’s XPath query stripped by all indexes.  
Example: indexless_xpath="/html/body/div/div"

shortened_indexless_xpath The shortened indexless XPath query is a combination of shortened_xpath and indexless_xpath.  
Example: indexless_xpath="/html/body/div"

regex_shortened_xpath Regular expression-based shortened XPath queries are even more flexible than simply shortened XPath queries. They have to be deployed carefully.  
Example: regex_shortened_xpath="/html[1]/body[1]/div[1]/div[2].*"

regex_indexless_xpath If the property regex_indexless_xpath is chosen, a regular expression of the item’s indexless XPath query is added.  
Example: regex_indexless_xpath="/html/body/div/div.*"

regex_shortened_indexless_xpath By deploying the regex_shortened_indexless_xpath, a regular expression is added representing an extendable version of shortened_indexless_xpath.  
Example: indexless_xpath="/html/body/div.*"

parent_xpath An item’s parent XPath query is the XPath query of the lowest ancestor of the given item.  
Example: parent_xpath="/html[1]/body[1]/div[1]"

maximum_ancestor_xpath The maximum_ancestor_xpath is the XPath query of the highest ancestor of the current item that does not contain additional text compared to the current item’s one.  
Example: maximum_ancestor_xpath="/html[1]/body[1]"

last_tag last_tag returns the part of the XPath query being located behind the last slash. It is stripped from its index.  
Example: last_tag="div"

last_tags last_tags is a more generic version of last_tag. It returns an arbitrary number of tags at the end of the item’s XPath query.  
Example: last_tags="div/div"

noisy An item is judged as noisy if one of the elements provided through a parameter is contained in the item’s XPath query.  
Example: noisy="false"
**Extraction Algorithm Properties**

**type**
The type of an item can be defined by the extraction plan designer. For the algorithm in this work, possible types are “list”, “group”, and “candidate”.
Example: `type="list"

**A.2.2. Purging Properties**
The purging step in the cluster algorithm allows cleaning up created clusters to improve their content quality. Only few properties have been implemented here since this step is less important.

- **maximize_xpath** This property instructs the clustering routine to iteratively remove the elements with the shortest XPath queries from a cluster until the left elements’ XPath queries reach a certain homogeneity or a minimum cluster size is reached.

- **sort** Cluster items may be sorted by various properties that can be provided with this function.

- **keep_biggest_items** The `keep_biggest_items` property removes the smallest cluster items until a certain size is reached.

- **split_by_maximum_xpath_levenshtein** Clusters may be split if the Levenshtein distance of contained elements surpasses a given threshold.

- **split_by_std_dev_text_length** This property allows splitting clusters if the standard deviation of text lengths for included items is greater than a certain threshold.

- **split_on_alternating_index** If a cluster contains item XPath queries with alternating indexes, the cluster can be split up into two new clusters with this property.

**A.2.3. Dropping Properties**
During the clustering process, many clusters may be created that follow the given clustering criteria but do not contain valuable information. For example, invisible clusters are not important for extracting product specifications. Therefore, all boolean clustering properties given above can be used for dropping clusters. Additional dropping properties are provided in the following.

- **has_varying_<measure>** A property `has_varying_width` checks if at least one item in the cluster does not have the same width as the others.

- **has_equal_x_distances** This property checks if all vertical neighbors in the cluster have the same distance to each other.

- **has_equal_y_distances** This property checks if all horizontal neighbors in the cluster have the same distance to each other.

- **has_varying_text** The property `has_varying_text` returns true if at least one cluster item does not include the same text as the others.
**has_alphanumeric_text**  Alphanumeric text should contain at least one letter and one number. If this is not the case, **has_alphanumeric_text** returns false.

**has_size**  The **has_size** property checks if the current cluster contains a certain number of elements.

**has_min_size**  If a cluster does not include a minimum number of elements given by a parameter, **has_min_size** returns false.

**has_constant_item_size**  The **has_constant_item_size** property returns true if all items in a cluster have the same size.

**is_not_noise**  A cluster is not noise if its elements are not noisy.
This appendix offers some additional screenshots for the Fedseeko web application which have not been included in the evaluation chapter. The screenshots display two different processes. The first process is a typical offers search which is executed with the help of the described plugin. All developed services take part in this process. Product specifications are presented in the producer’s terminology. The second process pictures a typical application flow for comparing products. The comparisons work on homogeneous sets of product specifications which have previously been transformed into Fedseeko’s terminology.

B.1. Offer Search

You may enter a query in the search form above. Subsequently, the query is categorized by the Offers Service in the background. Relevant offers are retrieved from selected shops and ordered by the relevance of your query’s category. By creating your own account Fedseeko can remember personal shop preferences.

Figure B.1.: Query Suggestions in the Offer Search View.
Figure B.2.: Cross-Site Request Plugin Interaction during Offers Page Retrieval.

Figure B.3.: Offers from Amazon, Ebay, and Evendi.com for “easyshare”.
Figure B.4.: Loading Detail View for Kodak Easyshare C195 Digital Camera (Purple).
Figure B.5.: Detail View including Preview on (unmatched) Product Specifications for Kodak Easyshare C195 Digital Camera (Purple).
B.2. Product Comparison

Figure B.6.: Settings Window of Cross-Site Request Plugin with Cursor Pointing on Previously Used Option.

Figure B.7.: Overview of Available Categories for Product Comparisons.
Figure B.8.: Available Facets for Digital Camera Category with two Breadcrumbs.
Figure B.9.: Comparison of Remaining Products for Chosen Facets.


