PROCESS-ORIENTED SEMANTIC WEB SEARCH
Studies on the Semantic Web

Semantic Web has grown into a mature field of research. Its methods find innovative applications on and off the World Wide Web. Its underlying technologies have significant impact on adjacent fields of research and on industrial applications. This new book series reports on the state of the art in foundations, methods, and applications of Semantic Web and its underlying technologies. It is a central forum for the communication of recent developments and comprises research monographs, textbooks and edited volumes on all topics related to the Semantic Web.

www.semantic-web-studies.net

Editor-in-Chief:
Pascal Hitzler

Editorial Board:
Fausto Giunchiglia, Carole Goble, Asunción Gómez Pérez, Frank van Harmelen, Riichiro Mizoguchi, Mark Musen, Daniel Schwabe, Steffen Staab, Rudi Studer

Volume 010 - Tran Duc Than, Process-oriented Semantic Web Search

Publications

Vol. 001 Stephan Grimm, Semantic Matchmaking with Nonmonotonic Description Logics

Vol. 002 Johanna Völker, Learning Expressive Ontologies

Vol. 003 Raúl García Castro, Benchmarking Semantic Web Technology

Vol. 004 Daniel Sonntag, Ontologies and Adaptivity in Dialogue for Question Answering

Vol. 005 Rui Zhang, Relation Based Access Control

Vol. 006 Jens Lehmann, Learning OWL Class Expressions (This book is also vol. XXII in the “Leipziger Beiträge zur Informatik” series)

Vol. 007 José Manuel Gómez-Pérez, Acquisition and Understanding of Process Knowledge using Problem Solving Methods

Vol. 008 Markus Krötzsch, Description Logic Rules

Vol. 009 Alexandre Passant, Semantic Web Technologies for Enterprise 2.0
Process-oriented Semantic Web
Search

Tran Duc Thanh
Institute AIFB, 76128 Karlsruhe, Germany
Preface

In this book, we elaborate on the concepts, objectives, problems, and challenges of searching on the Semantic Web generally called Semantic Web Search. Based on a general search model, we compare different concepts that exist in literature. Specifically, the concept of data retrieval can be distinguished from the one of document retrieval. Independent of the search tasks and the items to be retrieved, there is one paradigm called Semantic Search, which is centered on the use of semantics. Semantic Web Search is considered as a special kind of Semantic Search, which is focused on the task of data retrieval on the Semantic Web. The main objective of Semantic Web Search is to address more complex information needs. This breaks down to delivering more relevant results to more complex queries, which might range from precise answers in the form of facts, to complex results in the form of entities and their relations, up to integrated units of content that combine heterogeneous data from different sources on the Web. Challenges to this end include dealing with the large and increasing volume of data on the Semantic Web, its heterogeneity as well as its complexity. In fact, complexity in this scenario has many facets. From the system point of view, complexity poses additional requirements for data management and processing. But also, complexity imposes an additional burden on the user. An effective Semantic Web Search solution must be not only efficient, scalable and deliver high quality results but also, the users must be able to exploit it.

We discuss the state of the art of Semantic Web Search and in this context, position the specific contributions we made in this work. These contributions are integrated to form a holistic Process-oriented Semantic Web Search approach called SemSearchPro, one of the first solution towards large-scale Semantic Web Search, and the first one of its kind that does not entirely focus on the main search task of matching queries against system resources but actually considers the entire search process. While
SemSearchPro also aims at dealing with the main challenges of data volume, heterogeneity and complexity, it also considers the complexity users have to deal with. Its supports go beyond query processing to include additional steps of the search process – from query construction to query refinement up to result presentation and exploration – in order to facilitate users in dealing with complex information needs, queries, and results on the Semantic Web.

This book focuses on SemSearchPro and the research contributions made to realize this Process-oriented Semantic Web Search approach.
Acknowledgements

This work would not have been possible without the support and guidance of many people. First of all, I would like to thank my advisor Professor Dr. Rudi Studer for giving me the opportunity to do this research. Throughout my studies he granted me the freedom, the trust, and the help I needed.

I would like to thank the phenomenal team at the AIFB research group. In particular, I am grateful to Dr. Philipp Cimiano, Dr. Peter Haase, and Dr. Sebastian Rudolph for many fruitful advises, discussions and debates. Special thanks also to Günter Ladwig who has substantially contributed to the realization and implementation of my work. Last but not least, I would also like to thank Haofen Wang and other students from the Apex Lab for the successful collaboration throughout the last three years.

Above all, I am indebted to my friends, family, and to Tran Minh Ha and Tran Thi Van, the parents I love and respect.
## Contents

List of Figures ........................................... xii
List of Tables ............................................ xvii
List of Symbols ........................................... xix
List of Abbreviations ................................... xxi

**Chapter 1. Introduction**  ................................ 1
  1.1 Semantic Web .................................. 2
  1.2 Semantic Web Search ............................ 3
  1.3 Contribution of this Book ....................... 4
  1.4 Focus of this Book ............................... 8
  1.5 Organization of this Book ..................... 9

**Chapter 2. Semantic Web Search** ................. 13
  2.1 Introduction .................................. 13
  2.2 Search ........................................ 14
    2.2.1 Data Retrieval ............................ 17
    2.2.2 Document Retrieval ....................... 18
  2.3 Data on the Semantic Web ..................... 20
    2.3.1 Data and Metadata ......................... 20
    2.3.2 Data and Document Models ............... 23
    2.3.3 Semantics and Semantic Data ............ 25
    2.3.4 Ontologies ................................ 27
    2.3.5 Linked Data ................................ 27
    2.3.6 Embedded Semantic Data .................. 30
  2.4 Queries on the Semantic Web ................. 30
    2.4.1 Keyword Queries .......................... 31
    2.4.2 Conjunctive Queries ....................... 31
    2.4.3 SPARQL Queries ............................. 32
Chapter 3. The State of the Art of Semantic Web Search 41

3.1 Introduction 41
3.2 Objectives and Challenges 42
3.3 Crawling Semantic Data 46
3.4 Managing and Querying Semantic Data 49
  3.4.1 IR-based Approaches 49
  3.4.2 DB-based Approaches 52
  3.4.3 Native Approaches 54
3.5 Ranking Semantic Data 55
  3.5.1 Query-independent Ranking 55
  3.5.2 Query-dependent Ranking 56
3.6 Semantic Data Retrieval on the Web 57
  3.6.1 Federated Query Processing 59
  3.6.2 Data Integration 62
3.7 Conclusions 65

Chapter 4. Supporting the Semantic Web Search Process 67

4.1 Introduction 67
4.2 Data and Queries in SemSearchPro 69
  4.2.1 System Resource Model 70
  4.2.2 System Query Model 71
  4.2.3 Semantic Model 73
4.3 Process-oriented Schema-agnostic Search With SemSearchPro 76
  4.3.1 The Search Process in SemSearchPro 76
  4.3.2 Offline Pseudo-Schema Construction in SemSearchPro 78
  4.3.3 Query Construction in SemSearchPro 82
  4.3.4 Query Processing in SemSearchPro 86
  4.3.5 Result Presentation in SemSearchPro 89
  4.3.6 Query Refinement in SemSearchPro 92
4.4 Process-oriented Schema-agnostic Search Systems 92
  4.4.1 Close Environment: Single-Source Search in Semantic Wiki 94
### List of Figures

1.1 Topical structure of the book. 11

2.1 Topical structure of the Semantic Web Search chapter. 15
2.2 Linked open data cloud. 28
2.3 Extract of linked data on the Web. 29
2.4 Process-oriented search model. 37

3.1 Topical structure of the state of the art chapter. 43
3.2 Index structure for PosIdx. 51
3.3 Relation expansion using PosIdx. 52
3.4 (a) Example queries and the (b) Hermes infrastructure. 58
3.5 Query decomposed into three parts. 60

4.1 Topical structure of the SemSearchPro chapter. 69
4.2 A graph-structured resource model. 71
4.3 A graph-structured query model. 72
4.4 A semantic model. 74
4.5 Using semantic models for the search process. 77
4.6 A semantic model used for translation. 83
4.7 A query space. 85
4.8 Three paths through the query space and their scores. 86
4.9 An extended example of the resource model. 87
4.10 The semantic model as an answer space for the extended resource model shown in Fig. 4.9. 88
4.11 An extended example of the query graph. 89
4.12 Presentations of results and queries. 93
4.13 Effectiveness of search by query type. 98
4.14 Effectiveness of search by task. 99
4.15 Participants characterize the system, multiple answers were allowed. 99
4.16 Efficiency of search by query type. 100
4.17 Efficiency of search by task. 101
4.18 Efficiency measured by search-process steps taken. 102
4.19 Results of the usability study. 103

5.1 Topical structure of the query construction & refinement chapter. 106
5.2 The keyword-driven schema-agnostic search process. 111
5.3 The search process for the running example. 111
5.4 Example resource model. 120
5.5 (a) Single table schema and (b) example queries. 121
5.6 Data preprocessing and query computation. 122
5.7 Query space for the running example. 124
5.8 (a) Exploration on the query space and (c) query mapping. 124
5.9 MRRs of different scoring functions for DBLP. 145
5.10 MRRs of different scoring functions for TAP. 147
5.11 Query performance for DBLP data. 148
5.12 (a) Query performance and (b) index performance. 148
5.13 (a) Skills in using search engines and (b) experience with different technologies. 150
5.14 Percentage of unsolved tasks. 151
5.15 (a) Number of basic operations, (b) average time users needed to solve a task (in seconds). 153
5.16 (a) Most popular paradigm, (b) ease of choosing correct result (result completion) and (c) ease of choosing correct query (query completion). 154

6.1 Topical structure of the query processing chapter. 158
6.2 (a) A data graph, (b) a snippet of its semantic model graph (containing only edges with $l \in L_R$) and (c) a query graph for our example, where $u$ stands for undistinguished and $d$ stands for distinguished variable. 163
6.3 Example indexes for VP and SP. 169
6.4 The index match $h_{\sim_1}$. 171
6.5 Data graph matching for $q_{pruned}, h_{\sim_1}$. 171
6.6 An inverted index for the pattern EPSO. 190
6.7 Total times for (a) DBLP and (b) LUBM50; times for index matching, loading and joining for (c) DBLP and (d) LUBM50. 194
6.8 Performance for LUBM datasets. 196
6.9  Average SQP and OSQP times for all queries.  
6.10 Average SQP and OSQP times for some query types.  
6.11 Ratio of structure index size to data size.  
6.12 Times for different bisimulation depths.  

197 197 198 198
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Transforming semantic data to documents.</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>Intermediate result sets for $q_1$ and $q_2$.</td>
<td>61</td>
</tr>
<tr>
<td>3.3</td>
<td>Intermediate result sets for $q_2$ and $q_3$.</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>Task set 1 for semanticweb.org.</td>
<td>97</td>
</tr>
<tr>
<td>4.2</td>
<td>Task set 2 for semanticweb.org.</td>
<td>97</td>
</tr>
<tr>
<td>5.1</td>
<td>Query examples for DBLP.</td>
<td>144</td>
</tr>
<tr>
<td>5.2</td>
<td>Query examples for TAP.</td>
<td>146</td>
</tr>
<tr>
<td>5.3</td>
<td>Tasks of group A.</td>
<td>150</td>
</tr>
<tr>
<td>6.1</td>
<td>Triples for extension E2.</td>
<td>168</td>
</tr>
<tr>
<td>6.2</td>
<td>Statistics for the data graphs and structure indexes.</td>
<td>185</td>
</tr>
<tr>
<td>6.3</td>
<td>Single atom queries for the LUBM and DBLP datasets.</td>
<td>186</td>
</tr>
<tr>
<td>6.4</td>
<td>Path queries for the LUBM and DBLP datasets.</td>
<td>187</td>
</tr>
<tr>
<td>6.5</td>
<td>Star queries for the LUBM and DBLP datasets.</td>
<td>188</td>
</tr>
<tr>
<td>6.6</td>
<td>Entity queries for the LUBM and DBLP datasets.</td>
<td>189</td>
</tr>
<tr>
<td>6.7</td>
<td>Graph-structured queries for LUBM and DBLP datasets.</td>
<td>200</td>
</tr>
</tbody>
</table>
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Cursor representing paths.</td>
<td>132</td>
</tr>
<tr>
<td>$c.n$</td>
<td>Graph element $n$ just visited by cursor $c$.</td>
<td>132</td>
</tr>
<tr>
<td>$c.k$</td>
<td>Keyword element representing the origin of the cursor $c$.</td>
<td>132</td>
</tr>
<tr>
<td>$c.p$</td>
<td>Parent cursor of $c$.</td>
<td>132</td>
</tr>
<tr>
<td>$c.d$</td>
<td>The length of the path represented by $c$.</td>
<td>132</td>
</tr>
<tr>
<td>$c.w$</td>
<td>The weight / cost of the path represented by $c$.</td>
<td>132</td>
</tr>
<tr>
<td>$d_{\text{max}}$</td>
<td>The maximum radius of the neighborhoods.</td>
<td>132</td>
</tr>
<tr>
<td>$k$</td>
<td>The number of results to be computed during top-k.</td>
<td>132</td>
</tr>
<tr>
<td>$B$</td>
<td>A bisimulation.</td>
<td>166</td>
</tr>
<tr>
<td>$C()$</td>
<td>Function for computing cost of queries.</td>
<td>120</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Sorted list of cursors representing paths.</td>
<td>132</td>
</tr>
<tr>
<td>$E$</td>
<td>The edges of the data graph (resource model).</td>
<td>124</td>
</tr>
<tr>
<td>$E_R$</td>
<td>Like $E$, this one refers to the edges of the resource model.</td>
<td>70</td>
</tr>
<tr>
<td>$E_S$</td>
<td>The edges of the semantic model.</td>
<td>73</td>
</tr>
<tr>
<td>$E_Q$</td>
<td>The edges of the query graph.</td>
<td>72</td>
</tr>
<tr>
<td>$K$</td>
<td>The set of keywords. In the query construction part, it is overloaded to refer to keyword matching elements.</td>
<td>83</td>
</tr>
<tr>
<td>$L$</td>
<td>The labels of edges of the data graph.</td>
<td>70</td>
</tr>
<tr>
<td>$L_A$</td>
<td>The labels of attribute edges of the data graph.</td>
<td>70</td>
</tr>
<tr>
<td>$L_R$</td>
<td>The labels of relation edges of the data graph.</td>
<td>70</td>
</tr>
<tr>
<td>$L_Q$</td>
<td>The labels of edges in the query graph.</td>
<td>72</td>
</tr>
<tr>
<td>$LQ$</td>
<td>List of queues $Q$, each with cursors for exploration.</td>
<td>134</td>
</tr>
<tr>
<td>$M$</td>
<td>The matching framework used for search.</td>
<td>15</td>
</tr>
<tr>
<td>$N_K$</td>
<td>Graph elements that match the keywords in $K$.</td>
<td>83</td>
</tr>
<tr>
<td>$\mathcal{P}$</td>
<td>The presentation framework for process-oriented search.</td>
<td>36</td>
</tr>
<tr>
<td>$\mathcal{P}_R$</td>
<td>The framework used for presenting resources.</td>
<td>36</td>
</tr>
<tr>
<td>$\mathcal{P}_Q$</td>
<td>The framework used for presenting queries.</td>
<td>36</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>The system resource model, or data graph, respectively.</td>
<td>15</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{R}_S$</td>
<td>Also refers to the system resource model.</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{R}_P$</td>
<td>The presentation resource model.</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{R}_\sim$</td>
<td>The bisimulation-based semantic model also referred to as the structure index, or simply, the index graph.</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>The semantic model.</td>
<td></td>
</tr>
<tr>
<td>$S_Q$</td>
<td>The query space based on the semantic model.</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>The translation framework for process-oriented search.</td>
<td></td>
</tr>
<tr>
<td>$kCAN$</td>
<td>Top-k candidate results.</td>
<td></td>
</tr>
<tr>
<td>$kTOP$</td>
<td>Top-k results.</td>
<td></td>
</tr>
<tr>
<td>$V^R$</td>
<td>The nodes of the data graph.</td>
<td></td>
</tr>
<tr>
<td>$V^S$</td>
<td>The nodes of the semantic model.</td>
<td></td>
</tr>
<tr>
<td>$[v]$</td>
<td>A node of the structure index graph (semantic model).</td>
<td></td>
</tr>
<tr>
<td>$V^E$</td>
<td>The entity nodes of the data graph.</td>
<td></td>
</tr>
<tr>
<td>$V^V$</td>
<td>The data value nodes of the data graph.</td>
<td></td>
</tr>
<tr>
<td>$V^C$</td>
<td>The class nodes of the data graph / semantic model.</td>
<td></td>
</tr>
<tr>
<td>$V^D$</td>
<td>The data types nodes of the data graph / semantic model.</td>
<td></td>
</tr>
<tr>
<td>$V_{var}$</td>
<td>The variable nodes of a query graph.</td>
<td></td>
</tr>
<tr>
<td>$V_{con}$</td>
<td>The constant nodes of a query graph.</td>
<td></td>
</tr>
<tr>
<td>$Q$</td>
<td>The system query model, or query graph, respectively.</td>
<td></td>
</tr>
<tr>
<td>$Q_S$</td>
<td>Also refers to the system query model.</td>
<td></td>
</tr>
<tr>
<td>$Q_U$</td>
<td>The user query model.</td>
<td></td>
</tr>
<tr>
<td>$Q_P$</td>
<td>The presentation query model.</td>
<td></td>
</tr>
</tbody>
</table>
List of Abbreviations

AIS  Ascending Integer Stream ........................................... 51
DB   Database .......................................................... 34
IR   Information Retrieval .............................................. 2
OWL  Web Ontology Language ........................................... 2
LOD  Linking Open Data .................................................. 3
NL   Natural Language ................................................... 20
RDF  Resource Description Framework ................................. 2
RDFS RDF Schema .......................................................... 2
RR   Reciprocal Rank .................................................... 144
TA   Threshold Algorithm .............................................. 136
URI  Uniform Resource Identifier .................................... 27
URL  Uniform Resource Locator ....................................... 48
Chapter 1

Introduction

In the first chapter, this book motivates the Semantic Web Search topic by introducing readers to the vision of Semantic Web and trends. Section 1.1 discusses historical development and in particular, emphasizes recent trends of publishing and linking semantic data. The availability of semantic data in the form of resource descriptions on the Semantic Web today, and the large increase of this freely usable “Web commodity” are driving and will continue to boost research and commercial endeavors towards exploiting the Semantic Web. In this regard, searching the Semantic Web is a hot topic. Semantic Web Search solutions built for this purpose can support more complex information needs. They enable Web users to retrieve more relevant resources that are delivered in more precise and accessible presentation formats. To date, there exists a large body of research work on Semantic Web Search. Also, varying commercial solutions from major industry players such as Google and Yahoo, as well as from smaller but more innovative competitors such as Powerset and Hakia, have been positioned on the market.

The driving forces and concepts of Semantic Web Search will be briefly discussed in Section 1.2. In Section 1.3, we summarize the main contributions we made towards the realization of scalable, affordable and usable Semantic Web Search. In particular, we will point out in Section 1.4 that the focus of this book lies in the process-oriented perspective on Semantic Web Search. The organization of and suggestions for reading this book are finally presented in Section 1.5.
1.1. Semantic Web

Semantic Web is a field of research that is becoming more mature, as witnessed e.g. by the conference series “International Conference of Semantic Web”, which will celebrate its 9th anniversary in 2010. In the course of development, there is a vast body of resources that has been made available on the Semantic Web. These resources are specified using standard Semantic Web languages. Most commonly, resources are defined using the Resource Description Framework (RDF). The language for specifying the RDF vocabulary is called RDF Schema (RDFS). The other Semantic Web vocabulary language that is more expressive than RDFS is called the Web Ontology Language (OWL). We will now briefly introduce these Semantic Web standards.

- **RDF** This has become the standard language for modeling and representing resources on the Web. It is a general data model that allows for making statements about resources in the form of subject-predicate-object expressions also known as triples. These triples form a directed multi-graph such that an RDF data collection is intrinsically graph-structured. RDF is in fact a conceptual data model that has different serialization formats. Two widely used formats are RDF/XML and N3.

- **RDFS** This is an extensible knowledge representation language that provide constructs to specify the RDF vocabulary. The main constructs are `rdfs:Class` to declare that an RDF resource is a class for other resources, `rdfs:subClassOf` to define hierarchies of classes, `rdfs:Property` to specify that a resource is a property, `rdfs:domain` to represent the class of the subject as well as `rdfs:range` to represent the class of the object of the property.

- **OWL** This language includes many of the constructs that are available in RDFS. Additional constructs are provided to specify different kinds of complex classes. Since OWL essentially corresponds to the language of Description Logic, OWL statements have formal semantics. This can be exploited by reasoners to infer knowledge that is only implicitly entailed in the data. Since OWL is more expressive than RDFS, richer forms of knowledge can be inferred which however, come at the expense of scalability and performance. While the value of OWL has been demonstrated for specialized domains such as Medical Science, its practicability for the large-scale Web remains unclear and challenging.
On the Semantic Web today, there is quite a large number of ontologies available in RDFS and OWL. More recently, the trend of publishing and linking Semantic Web data coming from different sources has also resulted in large amounts of “free” data. Billions of RDF triples are now publicly available on the Semantic Web. To name a few prominent examples, the Linked Open Data\(^1\) (LOD) provides a large amount of datasets containing knowledge from different domains. The Semantic Web Health Care and Life Sciences Interest Group\(^2\) has actively promoted data publishing and integration and now, is able to provide a large amount of high quality domain specific data. Also, this data has been made available as linked data that is connected with data from other domains. Collectively, linked data published as part of the LOD project comprises hundreds of sources containing over 13.1 billion RDF triples, which are connected by around 142 millions of links (November 2009).

### 1.2. Semantic Web Search

Since the very beginning of the Web starting from the early 90s, the problem of finding resources has been at the center of research and commercial interests. The very first solution was based on manually created directories listing all websites in categories. However, the enormous growth of the Web resulted in an amount of documents that quickly become too large to be manageable in a manual way. Machines automatically crawling the Web and collecting information about websites have become common practice. Today, Google and other major players show that search engines scale to the Web, which has grown to more than 100 million documents\(^3\).

The predominant paradigm supported by these Web search providers is *keyword search*. It has become the most popular mean for finding information on the Web. Using simple keywords, users express their needs and Web search engines return a list of ranked documents that have been identified to be relevant.

In the light of the development of a Semantic Web comprising of ontologies and linked data as described previously, it is clear that there is much more potential to exploit than what is harnessed by Web search

\(^{1}\)http://linkeddata.org/

\(^{2}\)http://www.w3.org/blog/hcls

\(^{3}\)Assumed that every active domain has at least one website: http://www.domaintools.com/internet-statistics/, Oct 2008
engines today. Motivated by economic incentives, start-ups like Cuil\textsuperscript{4}, Hakia\textsuperscript{5}, and the bigger rivals like Microsoft and Yahoo challenge Google with new solutions and get widespread attention even in mainstream media in recent years. All challengers emphasize one claim: they are able to deliver more precise search results by means of Semantic Search. The term Semantic Search has various meanings, and is used in different contexts to refer to different features that exploit semantics. One central theme that underlies the new solutions is the use of Semantic Web resources. We use the term Semantic Web Search to refer to search solutions, which make use of the increasing amount of data on the Semantic Web.

Meanwhile, the exploitation of Semantic Web data (also referred to as semantic data) is a target for Yahoo but also Google. Semantic Web data embedded in Web pages are already used by these companies for richer presentation of Web search results. It is expected that through initiatives such as Google’s Rich Snippets and Yahoo’s Search Monkey, which actively promote site owners to publish and share high quality data, the amount of semantic data and Web resources associated with rich semantics will be drastically increasing on the Semantic Web. A virtuous cycle is about to be established where Semantic Web Search will be continuously driven by the mass availability of Semantic Web data and in turn, the development of Semantic Web Search will result in more data of higher quality. Just like Web documents, Semantic Web data will become Web commodities based on which a wider range of complex information needs can be addressed.

1.3. Contribution of this Book

Semantic Web data can help to address two principal types of information needs: (1) one is document retrieval where the goal of the user is to obtain documents and (2) the other is often referred to as data retrieval where the subjects of interest are pieces of data. This book focuses on the latter and partially deals with the former type of information needs. The problems that come along the way and the research contributions we made to address them can be summarized as follows:

- **Semantic-enabled Document Retrieval** Documents are mainly avail-

\textsuperscript{4}www.cuil.com

\textsuperscript{5}www.hakia.com
able as text. Correspondingly, models for representing documents that are used for document retrieval are primarily based on text, also called “bag-of-words” models. With the development of the Semantic Web, a large amount of semantic data can be found associated with or embedded in documents. The question that arises from that is how to exploit this data? In order to do that, the first step is to develop rich document models that can unfold the expressiveness of semantic data. We have developed such a model that uses the W3C recommended Semantic Web languages as well as de-facto metadata and vocabulary standards [TBCH07]. It is an ontology-based model that allows rich descriptions of documents to be expressed using semantic data in RDF and OWL. Using this model, document retrieval can be treated as a semantic data retrieval problem. The advantage of this approach is that it allows complex information needs to be specified and addressed, taking the richness of the structure and semantics in the data into account. In the application, an expressive document description is obtained by associating a document surrogate with possibly very complex pieces of semantic data, e.g. to assert that “document 101” describes “the deployment of a question answering system that has been developed by the Karlsruhe Institute of Technology” where the latter part is given as semantic data. The technical challenge behind this is thus the management of metadata, i.e., the type of data that refers to some possibly complex pieces of data. In collaboration with researchers from the University of Koblenz-Landau, we developed a framework and provided working implementations for dealing with metadata in RDF [SSST08]. It allows RDF statements to be made about complex sets of other RDF statements. These statements can be queried and combined in various ways. Also, we developed a framework for dealing with metadata in OWL [THM+08]. The problem with OWL is that its higher expressiveness might result in situations where reasoners employed to process the data, make uncontrolled inferences that are not desirable when meta-level statements interact with the other statements. The framework we developed allows for a natural separation of these two levels. Also, it contains a powerful query mechanism to retrieve statements at the two levels as well as to combine them.

- **Semantic Data Management** The exploitation of data on the Semantic Web requires, for search and other applications alike, con-
cepts and mechanisms for semantic data management. The data has to be physically organized on disk in an appropriate way and indexed to allow for fast access. Due to the rich structure and semantics, querying the data is a hard task. Scaling this to the large amount of semantic data available requires efficient procedures for join processing, join order optimization and mechanisms for ranking to focus on the best results that are relevant to the user. To this end, we have focused on two aspects: (1) the use of Information Retrieval (IR) technologies for storing and indexing a large amount of data and (1) the adoption and extension of database techniques for the efficient physical organization of semantic data and processing of complex queries – both in a single- and the truly multi-source scenario on the Web. In particular, in collaboration with research colleagues from the Apex Lab of the Shanghai Jiao Tong University, we proposed the use of the inverted index that is commonly employed in IR, for storing and retrieving semantic data [WLP+09]. Further work we did along this line was on supporting efficient updates when using such an index to deal with semantic data [LWL+08]. In [TL10], we proposed the use of structure-based partitioning and a procedure for structure-aware query processing. The idea of the former is to physically group data elements that are similar in structures. This way, groups corresponding to a given query contain more elements that satisfy the structure of the query. In other words, this partitioning can help to reduce IO by avoiding the loading of non-relevant data during query processing. The idea behind the latter concept is to use a structural summary during query processing to prune away certain parts of the queries. We showed in experiments that these two concepts are valuable when dealing with complex queries. Finally, we contributed also to the problem of multi-source search where results might combine data from different sources. For this, we designed a procedure for federated query processing which leverages precomputed mappings between sources to combine results in a more efficient way [TWH09].

- **Process-oriented Semantic Search** Search is centered on the task of matching queries against a collection of data. Accordingly, existing literature is focused on using semantic data as well as semantic models of varying expressiveness for solving this matching problem. Clearly, search is a more complex process that involves additional steps such as query formulation, query refinement and result
presentation. The rich structure and semantics in semantic data can be used not only for matching but also for the other steps involved in this process. In [TCRS07], we proposed the use of domain knowledge available in ontologies to interpret user keyword queries, i.e., to translate them to richer structured queries. This facilitates query formulation as the user can use simple keywords instead of formal structured queries. Further work we did along this line is to improve the efficiency by using a summary model of the data [WZL+08] and a top-k search procedure [TWRC09] to compute only the best ranked queries. For ranking the structured queries computed from user keywords, we proposed to use metrics based on the TFIDF concept commonly used in IR as well as additional metrics that take the authority of data elements as well as structure information given in the data into account [TWRC09]. In [TMH10], we present a theoretical analysis and experimental study of different approaches that can solve complex information needs without requiring the users to know the schema. This line of work illustrates the use of semantics for query construction. In [THS09], we presented a general process-oriented model of Semantic Search, demonstrating that besides for query construction as discussed above, semantics can be used to determine the right facets for users to browse and refine results. Also, a semantic model provides the necessary information to automatically select result-specific presentation elements. In short, it can be exploited to compute meaningful presentations for both queries and results. Further, the summary model we used for translating keywords to structured queries as well as the structural summary we proposed for structure-aware query processing can be seen as different kinds of semantic model. In this sense, the proposed approaches actually leverage semantics for more efficient query processing. The work we presented in [THS09] provides this big picture of using semantics throughout the search process – from query construction to query processing to result presentation and refinement.

• For all the contributions presented above, we provided working demonstrators and experimental results. We would like to highlight two system implementations that successfully integrate the research work we did: (1) In collaboration with colleagues from Apex Lab, we developed a system called Hermes that won the second prize at the 2008 Semantic Web Challenge. It integrates the work
on IR-based storage and indexing of semantic data [WLP+09], the query translation approach presented in [TWRC09] and the federated query processing mechanism [TWH09]. (2) Together with colleagues from the FluidOps Cooperation, we developed one another system called The Information Workbench. Besides query translation and query processing on structured data, this system is capable of supporting queries on textual data as well as hybrid queries on both structured and textual data. For this, the work we did on modeling documents as expressive meta-level descriptions of entities and events presented in [TBCH07] came into play. The Information Workbench became finalist of the 2009 Semantic Web Challenge and is now actively used and commercially exploited by FluidOps.

1.4. Focus of this Book

The book is devoted to the Process-oriented Semantic Search topic. In particular, it deals with the use of semantics throughout the process of retrieving resources on the Semantic Web – hence the title Process-oriented Semantic Web Search.

Since the amount of data on the Semantic Web that refers to or is embedded in documents is sufficiently large, this book partially covers the topic of document retrieval. It discusses the concept of search in general and breaks it down into the notions of document retrieval and data retrieval. In this context, the use of semantic data is discussed. In particular, we briefly present our contributions to semantic-enabled document retrieval by introducing the approaches for managing semantic metadata in RDF [SSST08] and OWL [THM+08] and the expressive ontology-based document model that is built upon this body of work [TBCH07].

More emphasize is put to the topic of semantic data retrieval. This is because still, most resources on the Semantic Web is given as data. Correspondingly, existing work on Semantic Web Search so far is dedicated to the problem of semantic data retrieval. This book presents an overview of the state of the art of Semantic Web Search. It shows that besides specific mechanisms for querying and ranking, Semantic Web Search engines rely on robust and scalable backends for semantic data management. The Semantic Web Search survey part of this book discusses different approaches that have been proposed for dealing with this problem. In this context, we briefly sketch and position the proposals we made for indexing and storing semantic data using inverted indexes [WLP+09, LWL+08], for structure-
aware partitioning and query processing [TL10] and for federated query processing [TWH09].

The focus of this book lies in providing an *process-oriented perspective on Semantic Web Search*. To this end, we present our Process-oriented Semantic Search model [THS09] and illustrate on the basis of concrete approaches and search applications that semantics can be exploited successfully throughout the search process. We show how it can be used for the presentation and refinement of queries and results. In more details, we discuss how it enables efficient translation of keyword queries and processing of structured queries.

1.5. Organization of this Book

According to this topical focus, the main part of the book is devoted to SemSearchPro, the approach we proposed for Process-oriented Semantic Web Search. Fig. 1.1 shows the topics and overall structure of the book.

In Chapter 2, we first introduce the *general search model* and break it down to data retrieval and document retrieval. According to this model, search as a concept consists of three principal components, namely the query representation, the resource representation and the matching framework. Based on this model, *Semantic Search* is introduced as a specialized search concept, which involves the use of a semantic model. *Semantic Web Search* is presented as a kind of Semantic Search that is mainly concerned with the retrieval of data on the Semantic Web. Last but not least, we discuss the concept of *Process-oriented Semantic Search*, which extends the general search model to arrive at a broader perspective on search. Search as a process includes besides the task of matching also the the steps of query construction as well as result presentation and refinement.

In Chapter 3, we begin with discussing the main objectives and challenges of *Semantic Web Search*. They are related with the volume and heterogeneity of the available semantic data and the complexity of information needs and queries, respectively, that shall be supported. We present an overview of existing Semantic Web Search engines. The inner working of these engines is analyzed and their features are mapped to the main classes of semantic data crawling, storage and indexing, query processing and query and result ranking. Also, single-source search is distinguished from search in the Semantic Web scenario, which requires dealing with multiple sources – to process queries against several data sources and in particular, to combine data from several sources to produce
integrated results. Concepts and techniques that have been proposed for these main tasks so far, are surveyed. Also, our own approaches targeting these problems of data management and search are briefly presented and compared against existing work. In the conclusion part, we recapitulate the current status of Semantic Web Search by analyzing the progress along the dimensions introduced in the beginning, namely data volume, data heterogeneity and query complexity.

In Chapter 4, we introduce our integrated search approach we coined SemSearchPro. It implements the Process-oriented Semantic Search model by using semantics throughout the steps of query construction, processing as well as result presentation and refinement. SemSearchPro particularly addresses the problems of semantic model management. On the Semantic Web, relying on an explicitly given semantic model is not always possible. In many cases, such a model is incomplete or simply not available. Even when it exists, the manageability of this model poses a problem because the underlying data quickly changes. Further problems are tractability and scalability. Highly formal and expressive semantic models enable logical reasoning, a feature that can be valuable for searching the Semantic Web. However, reasoning is a hard task. There exists yet no solution that can scale and makes this feature affordable for the Web scenario. SemSearchPro tackles these problems by employing a lightweight semantic model that can be automatically derived from the data. It shows that the semantics captured by this model is yet valuable, contributing to different steps of the search process. Most importantly, it can be processed on the large scale. Another property of the employed model is that it is extensible. This is important for the specific scenarios where the use of additional semantic constructs is valuable and affordable.

In Chapter 5, we provide details on the tasks of query construction and refinement. Specifying complex information needs is a difficult task especially for the lay Web users. Among many other requirements, the users need to know the schema. In this chapter, we elaborate on one particular class of approaches called keyword-driven schema-agnostic search. It is based on the keyword search paradigm that has become popular on the Web. The approaches discussed in the survey go beyond this simple keyword paradigm. In particular, we focus on keyword search in combination with faceted browsing and keyword search in combination with keyword interpretation. In this sense, interpretation refers to approaches that translate keywords to structured formal queries or map them directly to possibly complex structured results. Finally, this chapter presents the
SemSearchPro query construction approach that is based on the idea of computing structured queries from user keywords.

Chapter 6 elaborates on the task of query processing. The focus lies on the particular class of approaches that does not rely on an explicitly given schema – called schema-agnostic query processing approaches. That is, the goal is to support complex queries even when the data is not associated with a schema. We survey approaches that compute a pseudo-schema from the data, also called structure index. The SemSearchPro query processing approach is elaborated in detail. It is based on the use of a particular structure index that can be computed for general graph-structured data. SemSearchPro leverages this index to implement the concepts of structure-based data partitioning and structure-aware query processing.

We try to make every chapter of this book as self-contained as possible. Thus, every chapter can be considered as an independent piece that can be read in isolation. However, to reduce redundancy, some chapters of the book refer to the search concepts presented in Chapter 2 and the models employed by SemSearchPro presented in Chapter 4. Thus, when additional details are of interests, we suggest the readers to follow the explicit references that we have provided in the texts. Also, we use consistent notation for all models and elements throughout the book. Since the number of symbols is too large to be memorized, we suggest to always have the provided list of notation alongside when reading the text.
Chapter 2

Semantic Web Search

2.1. Introduction

This second chapter of the book discusses the concept of Semantic Web Search. The overall structure of this chapter is illustrated in Fig. 2.1.

Search is a broad term that is used in different contexts. The precise meaning is often not clear and in fact, there exist different conceptualizations. Based on a general model for search, different search paradigms will be discussed in Section 2.2. In particular, the concept of data retrieval will be compared with that of document retrieval.

Semantic Web data encompasses different kinds of things such as descriptions, services, people and products. It can also represent Web documents. Thus, searching the Semantic Web can be considered as a data retrieval task. For defining this task more precisely, Section 2.3 elaborates on the different types of data that is available on the Semantic Web today. This survey starts with the taxonomy of data that can be found on the Semantic Web. It makes clear that in fact, data on the Semantic Web, what is commonly referred to as semantic data, might be associated with a schema or more often, comes in isolation. In other words, semantic data might be fully-structured or semi-structured. The survey includes ontologies which represent the main artifacts of Semantic Web applications. Additionally, it discusses publicly available RDF data such as linked data promoted by the LOD project as well as data embedded in Web documents such as RDFa. Besides this overview on the different types of data, Section 2.3 also contrasts data with the notion of metadata. Complex document descriptions can be given as metadata. Based on such descriptions, this section shows that document retrieval can also be understood as a data retrieval task.
Last but not least, this section introduces various kinds of semantic models employed by different communities. This is to clarify what the notions of “semantics” and “semantic data” entail.

In Section 2.4, the query formalisms commonly used for searching Semantic Web resources are introduced. In particular, this survey includes the keyword queries that are commonly used by Web search engines, the conjunctive queries that represent an important fragment of various query languages, and the SPARQL queries that constitute the principal mean for retrieving RDF data.

Subsequently, this chapter provides in Section 2.5 precise definitions for two concrete Semantic Search models that involve the use of semantics, namely the basic Semantic Search model and the Process-oriented Semantic Search model. The latter model is the one emphasizing the process-oriented view on Semantic Search.

Section 2.6 provides a discussion on the subject of interest of this book, namely the problem of Semantic Web Search. It makes clear that Semantic Web Search is in fact about retrieving data on the Semantic Web. It can be seen is one particular type of Semantic Search.

This chapter concludes in Section 2.7.

2.2. Search

In our context, search can be conceived as an IT-supported process that aims at reducing what has been called the “information overload”. There exists a large number of resources, e.g. on the Web, on the intranet and even in the personal sphere. In total, the amount of information facing the user is beyond manageability. Search as an automated process helps the user effectively finding the right information, given the information needs. Typically, information needs are expressed in the form of queries, which are evaluated by the search engine and the “matching” information are returned back to the user as answers.

Beyond this general notion of search, there exists a wide range of more specific conceptualizations. Searching information has always been considered as a major challenge that drives IT research in various fields. In fact, as a research topic, search lies at the heart of the core communities of database and IR, and many more application oriented communities including the Semantic Web. As a result, various approaches for different kinds of search including information retrieval, document retrieval, text retrieval, data retrieval and Semantic Search have been proposed. Es-
sentially, they are dealing with the same problem. They all satisfy the properties of a basic search model that we define as follows:

**Definition 1 (Basic Search Model)** The basic search model is a tuple \( (\mathcal{R}, \mathcal{Q}, \mathcal{M}(\mathcal{Q}, \mathcal{R})) \) where

1. \( \mathcal{R} \) is the resource model, which is a set of syntactic representations for the underlying resources.
2. \( \mathcal{Q} \) is the query model, which is a set of syntactic elements representing the user information needs.
3. \( \mathcal{M}(\mathcal{Q}, \mathcal{R}) \) is the matching framework which models the relationship between resource representation and query representation. It specifies the notion of matching to compute whether the resource representation contains an answer to the query representation. In particular, there is a matching function \( \mathcal{M} : \mathcal{Q} \times \mathcal{R} \mapsto [0, 1] \), which for a given query representation \( q \in \mathcal{Q} \) and a resource representation \( r \in \mathcal{R} \), outputs the degree to which \( r \) matches \( q \).
While the different notions of search we encountered in existing literature can be described using this general model, they in fact vary in these three main components. That is, they assume different models of the information need, deal with different types of resources and results and in particular, use different techniques for matching queries against resources to produce results.

Clearly, the term resources used here shall be understood in a general context. It might stand for any objects that can be encountered in the real world, including physical being such as people, companies and books as well as those that exist only virtually, like computer services, applications and documents. Results that are returned to the users however, are not objects but object surrogates or as defined in the model, some syntactic representations of the objects. That is, the system does not return the objects but data representing those.

*Data retrieval* can be understood as a general discipline that deals with retrieving any kinds of objects that can be surrogated by data. In practice, data retrieval is often used to refer to database concepts that focus on the retrieval of *structured data* kept in a database. This data comes as tuples, which represent objects and possibly, complex relationships between them. Document retrieval, often also referred to as text retrieval, deals with a particular kind of object – namely documents. As opposed to data retrieval, document retrieval in classic terms, deals with *textual data* instead of structured data. Information retrieval can be conceived as a general term that encompasses both data and document retrieval. However, since the community behind it – the IR community – is mainly concerned with the problem of retrieving documents, it is often associated with the problem of document retrieval. Semantic Search is a term recently coined to put emphasize on the use of semantics for different kinds of retrieval tasks. We will discuss this concept in more details in Section 2.5.

Thus, with some overlaps, the usage of the terms data retrieval and document retrieval basically refers to the underling subjects of interest, which on the one hand, are real-world objects represented as structured data and documents given as textual data on the other hand. Due to this crucial difference, two separate bodies of theory, practices, and technologies have evolved. We will now provide a brief introduction to this two major retrieval paradigms.
2.2.1. Data Retrieval

To clarify the specific differences between these two retrieval paradigms, we decompose a query model into two parts: essentially, a query can be conceived as (1) a set of structure constraints and (2) one other set of content constraints (or textual constraints). The structure part specifies the structural conditions the answers must satisfy. The content part describes the content of the answer. An information need is for instance:

Example 1  Find employees working at KIT who used to work for IBM and have published work related to search and ontologies.

Clearly, the results to be expected here consist of real-world people. The structure constraints these people must satisfy are “used to work for organization x” and “published some work y”. Further, specific content-related constraints limit the candidates for x and y. Namely, x shall be something with the content IBM and the content of y can be described by the terms search and ontologies.

In a classic data retrieval scenario, the resource model \( \mathcal{R} \) is basically a set of tuples which basically, capture entities, their attribute values and their relations to other entities. When considering entities and values as nodes and attributes and relations as edges, these data elements constitute structures that might be as complex as general graphs. Data in relational databases and data in RDF stores are two examples of graph-structured data.

The major problem in data retrieval is how to process the structure constraints against the data to find structural matches. As this might be as complex a graph-pattern matching, which is NP-complete, a vast body of database research has been devoted to the study of processing techniques that can efficiently address this problem. The range of solutions encompass different aspects. The two main aspects that drive the efficiency of this process are basically (a) the access to the data and (b) the actual processing of the data to find matches to the query.

Data access is a problem because in most of the use cases, the amount of data to be managed is too large to be held in memory. It has to be stored on disk, and fetching data from disk is inherently slow and thus, often represents the major bottleneck in data retrieval. In any case, an efficient organization of the data is crucial to the performance. In-memory data organization is one topic. More difficult is the problem of on-disk data organization. There are simply more ways of how to physically par-
tion the data and how to store it on disk. Different types of indexes can be created to improve access times but on the other hand, this increases the needs for storage space. In short, there are different design alternatives concerning data partitioning, physical storage as well as indexing. The long tradition of database research has brought about a wide range of solutions along these dimensions.

Processing the queries against the data or more specifically, checking the structure constraints, is conducted primarily by means of join processing. The execution of joins lies at the heart of data retrieval. For this, different kinds of join implementation exist. In combination with the right index access and data fetching strategies, an efficient join implementation can greatly accelerate query processing. Beyond the execution of the individual joins, the body of research on query optimization deals with the problem of finding the optimal order of execution – of index and data access as well as join operators. In practice, the optimized processing of queries on the basis of existing database concepts and techniques greatly improves performance. The problem of query processing which might be NP-complete in worst case, can be solved at runtime in times that are often only linear to the size of the data.

To sum up, we provide a refinement of the general definition of search presented above to arrive at our definition of data retrieval:

**Definition 2 (Data Retrieval Model)** The data retrieval model is a tuple $\langle R, Q, M(Q, R) \rangle$ where

1. the resource model $R$ comprises of structured data.
2. The query model $Q$ is a set of structure constraints defined on the results using a structured query language.
3. $M(Q, R)$ is the framework for matching the structure constraints $Q$ against the structured data $R$. In particular, the matching function $\mathcal{M} : Q \times R \mapsto \{0, 1\}$ outputs whether resource in $R$ is a result to a query in $Q$ or not.

For this data retrieval problem, database research has produced data models, query languages and in particular, efficient solutions for implementing the matching framework.

### 2.2.2. Document Retrieval

While data retrieval is primarily concerned with the structure constraints of the user queries, the vast body of IR research and document retrieval
has been devoted to processing the content constraints. Given terms such as KIT and IBM, there are many documents or generally speaking, resources with content that match these terms. The problem to be addressed here is less the aspect of efficiency but effectiveness. In particular, the aim is to find the resources that most likely match user queries most often represented as simple lists of terms. While the matching function presented for data retrieval above is crisp, the one employed in document retrieval is fuzzy. In other words, data retrieval involves precise structure matching while document retrieval is based on an imprecise matching of query terms against the resources’ textual content. Thus, there is the underlying assumption that a given query does not uniquely identify the resources in a collection. Instead, several resources may match the query, perhaps with different degrees. Hence, research on document retrieval is very much centered on developing robust and effective mechanisms for ranking different results. More precisely, document retrieval can be defined as follows:

**Definition 3 (Document Retrieval Model)** The document retrieval model is a tuple \( \langle R, Q, M(Q, R) \rangle \) where

1. the resource model \( R \) comprises of textual representations.
2. The query model \( Q \) is a set of content constraints defined on the results using natural language or are simply given as a set of terms.
3. \( M(Q, R) \) is the framework for matching the content constraints \( Q \) against the textual representation \( R \). In particular, the function \( M : Q \times R \mapsto [0, 1] \) outputs the degree to which a representation in \( R \) is a result to a query in \( Q \).

The various solutions proposed for dealing with the document retrieval problem mainly vary in the representation of resources and queries \( R \) and \( Q \) and the matching framework \( M(Q, R) \). The two family of models that have become popular are (1) the set-theoretic models based on the standard boolean model and (2) general algebraic models that are built upon the vector space model. Whereas the former represents the textual content of documents and queries as sets of words or phrases the latter captures documents and queries as vectors and matrices. The boolean model does not support computing varying degrees of matching. This is possible with the second family of models, where the degree of matching between a query and a document is often computed as a scalar value of the underlying vectors.
The state of the art in matching and ranking is based on the application of statistical learning. One family of model that has dictated the scene is the probabilistic relevance framework and its derivatives, namely the binary independence model [JWR00b, JWR00a] and the okapi (BM25) relevance function [RWHB+92]. Recently, the family of language models [PC98] have attracted interests of IR researchers. Underlying all these approaches is the basic technique of probabilistic inference. The degree of matching is computed in terms of the probabilities that a document is relevant for a given query. The inference to be made for this is based on the application of the Bayes’ theorem. It is applied to exploit prior information about the distribution of terms and about relevance. Basically, probabilistic models are trained towards estimating the probability of a relevant document, given a query, and most importantly, given prior knowledge about relevance between queries and documents. At runtime, Bayesian inference is used to update the prior knowledge about relevance with new information to arrive at a more refined relevance estimate. As opposed to that, there is no explicit concept of relevance in the language modeling approach. The probability that a document is relevant is assumed to be proportional to the probability of generating the query using language models of documents. So, instead of directly estimating relevance, the goal is to compute the best language model for a given query.

2.3. Data on the Semantic Web

In this section, we discuss the notions of data and metadata and introduce the work we did on managing metadata in RDF and OWL.

2.3.1. Data and Metadata

2.3.1.1. Data

The principal component of the search models presented previously is data. Data might come in different formats, embodied in different objects etc. The distinction that is important for this book is as follows: (a) there are structured data versus (2) unstructured data also called textual data. When every data element is conceived as a node and connections between them are considered as edges, then elements of a structured data collection form structures that might be as complex as general graphs. As opposed to that, elements of an unstructured data collection form flat structures, i.e., simple list of terms that are not connected. Clearly, terms in natural
language (NL) and even in simple keyword queries bear positional information. However, this and the proximity information that can be derived from that are not considered as connections. The structures considered here are made up of explicit connections that explicitly associate one data element with one another.

2.3.1.2. Metadata

One important concept that often appears in the search context is metadata. Generally speaking, metadata is data about data. Recall that data is basically a representation of real-world objects. So, every data element stands for some particular object or set of objects. Structured data might describe any kinds of objects while textual data mostly stands for the content of documents (or some other resources). In other words, data represents object descriptions or surrogates. In this regard, metadata can be simply conceived as additional information about descriptions of objects (meta-level information). Examples of metadata are (1) the author of document ID01 is Rudi and (2) the creator of the tuple asserting that Thanh worked at IBM is Denny.

2.3.1.3. Metadata in RDF

Referring to single data elements like document ID01 in the first example is straightforward. In fact, most languages for data modeling and knowledge representation including the Entity Relationship model, RDF as well as OWL support this. Asserting meta-level information about complex data elements like in the second example raises more technical challenges. In the first case, the data element has an identifier (ID01). This can be used to refer to that element, e.g. to assert the statement \texttt{author}(Rudi, ID01).

In the second case, there is no such identifier for the complex data element capturing the fact \texttt{workedAt}(Thanh, IBM). Special mechanisms need to be provided to model this. One well-known technique also used in RDF for asserting meta-level statements is reification. To reify a complex data element basically amounts to create an identifier for that element.

Given such modeling primitives, how can we design querying formalisms to retrieve both data and metadata? How can metadata that covers different aspects of the underlying object descriptions be combined? For instance, metadata might describe the origins, authorship, recency or certainty of data, to name a few aspects. In the work presented in [SSST08], we developed a generic approach for managing many dimensions of metadata and provided the formalism as well as an implementation for query-
ing and combining them. The approach reuses existing RDF capabilities (i.e., reification support) in order to represent metadata. Then, it extends SPARQL query processing in such a way that given a SPARQL query for retrieving data, one may request metadata without modifying the query proper. Thus, this approach achieves highly flexible and automatically coordinated querying for data as well as metadata, while completely separating these two areas of concern.

### 2.3.1.4. Metadata in OWL

The problem of associating meta-level information with complex pieces of data exacerbates in semantically more expressive languages like OWL. The formal semantics of this language is typically exploited to perform reasoning, i.e., to infer additional information from the one explicitly given. The semantics of metadata may interact with the semantics of the actual data. This has impact on reasoning and might lead to inferences that are not desirable. Further, this interaction substantially increases the complexity of reasoning. In the work in [THM+08], we proposed a solution to this problem of metadata management in OWL. We presented a simple yet semantically sound framework for the management of metalevel information – one that can easily be integrated into existing ontology management systems and reasoners. Our framework is based on the observation that domain (data) and metalevel information (metadata) have distinct universes of discourse. We store the metalevel statements in the domain ontology using axiom annotations – ontology statements that are akin to comments in a programming language and that do not affect the semantics of the domain information. We give semantics to this information by translating the metalevel statements from the domain ontology into a metaview – an ontology that explicitly talks about the facts in the domain ontology. The domain ontology and its metaview are interpreted independently. Also, we proposed a query language called MQL that can be used to integrate the information in the two ontologies in a controlled manner.

We will now discuss why the modeling and management of metadata is relevant for search, especially for document retrieval.
2.3.2. Data and Document Models

2.3.2.1. Data Models

Data models formally define data elements and relationships among data elements for a domain of interest. Often, a data model can play the role of an external model (also called view), a conceptual model, or a physical model. A conceptual data model is also referred to as the data schema. It might capture flat, hierarchical, relational, network- or graph-structured data. For instance, an Entity Relationship diagram models relational data and ontologies in RDF and OWL contain graph-structured data.

2.3.2.2. Document Models

A document model is one particular data model that is concerned with documents. One well-known document model is the Document Object Model [WS97] that is widely used on the Web. It is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML. One aspect that is of particular interest in this book is that a document model specifies the primitives that can be used to capture documents. That is, it contains the primitives to specify the document representation $R$ of our document retrieval model.

Clearly, the aspect that is most interesting in the context of document retrieval is the document content. After all, it is the content that constitutes the basis for computing the degree of matching between queries and documents. As discussed previously, the content is represented in terms of textual data. Accordingly, the basic document model (bag-of-words model) that covers the document content only is purely based on textual data. For matching queries against textual data, we have discussed the body of IR research on probabilistic and language models. Apart from the textual content, a document model might also contain structured data. This is to accommodate scenarios where documents are associated with structured metadata such as the author, the publisher etc. This data can be exploited to improve document retrieval. Whereas IR methods can be used to deal with the textual data, database research provides the techniques to process structured data. Thus, an effective approach to document retrieval might combine standard document retrieval and data retrieval techniques discussed in Section 2.2 to match the query against a hybrid document model that comprises both textual and structured data.
2.3.2.3. Expressive Document Models for Ontology-based Document Retrieval

In the work in [TBCH07], we elaborated on one extreme solution that models document retrieval as a data retrieval problem. For this, a document model purely based on structured data is employed. In particular, we proposed an expressive ontology-based document model for representing documents in terms of elements of a domain ontology. This resource model is based on the use of RDF and when additional expressiveness is desired, also OWL. Further, established ontologies and metadata standards such as SUMO, MPEG-7 and Dublin Core have been incorporated to provide a reference model for ontology-based document retrieval.

Based on this model, documents can be represented in terms of RDF statements as well as complex OWL axioms. In particular, it distinguishes the document content from the physical object that bears it. Different kinds of metadata related to these two entities can be specified. For instance, the subject, the topic, the author can be specified for the content and size, format and publisher can be defined for the corresponding content bearing object. The expressiveness of the model lies in the explicit modeling of the content. Specifically, the subject could be any complex pieces of structured data. For instance, it is possible to specify that the subject of a content is about “a question answering system deployed at British Telecom”. Note that this requires associating meta-level information (i.e., “the content is about”) with data (“a question answering system...”). For this, we proposed to use the mechanisms for managing and querying metadata we have developed for RDF and OWL discussed above [SSST08, THM+08].

Based on this document model, the user can directly specify his information need at an enhanced level of expressiveness. In particular, it does not restrict the description to keywords but allows for the rich constraints specified in terms of complex descriptions that might involve resources, events and complex situations modeled in a domain ontology. We have shown that with the proposed model, a large set of retrieval functionalities can be supported to address complex information needs – needs that can not be expressed using keywords.

Summing up, document retrieval can be performed not only using IR methods but when considering metadata, might also involve database techniques. Given expressive document representations that are purely based on structured data, document retrieval can also be conceived as a data retrieval problem. In this case, database techniques as discussed be-
fore in Section 2.2 can be applied to search for documents.

2.3.3. Semantics and Semantic Data

The term “semantic” has the tendency to become a buzzword as many applications claim to be “semantic” or to feature “semantic search”. There exist many different conceptions and definitions for Semantic Search [GMM03, ZYZ+05, CCPC+06]. For instance, Semantic Search from the IR point of view [CFV07] is different from the understanding in the Semantic Web community. However, central to all Semantic Search approaches proposed so far is the use of a semantic model. In the following, we discuss several kinds of models that capture semantics in different ways. Based on this, we formally define a basic model for Semantic Search, which is sufficiently general to capture the main directions of Semantic Search. Then, we follow the direction of Semantic Search suggested by Hildebrand [HvOH07] to provide a formal model of Process-oriented Semantic Search.

2.3.3.1. Semantic Data Models

Generally, semantics is concerned with the meaning of things. Meaning is established through a semantic data model (or semantic model in short), which commonly captures interrelationships between elements and their interpretations. Various semantic models have been proposed and used in different research communities.

- There are linguistic models such as thesauri that capture relations between syntactic elements.
- In the database community, conceptual models such as Entity Relationship diagrams are used to capture relations between entities [Che76]. Thus, while linguistic models are concerned with meanings at the level of words, conceptual data models more specifically deal with meanings at the level of real-world entities denoted by words.
- There are also formal conceptual models based on languages of logic where interpretations are precise and computable [SS04].

In this book, semantic models refer to the family of models capturing the conceptualization but not the actual instantiation – as opposed to the actual data. This family of models includes linguistic models, conceptual models as well as formal models varying in the degree of expressiveness and formality. We now provide a general working definition:
Definition 4 (Semantic Models) Semantic models represent the family of models providing a conceptualization of the universe of discourse. The semantics of a model is established through interpretation, which is a mapping of syntactic elements to real world entities and their relations. A semantic model is expressive when there are many types of modeling constructs that can be used to specify syntactic elements of different semantics. A semantic model is formal when there is a defined interpretation function \( I \) based on which the interpretations of syntactic elements become computable.

Note that according to this definition, semantics is not always computable but might have to be established manually, i.e., the interpretation of the model has to be performed by the human worker. This is particular the case with linguistic models.

2.3.3.2. Semantic Data

As opposed to a data model or a semantic model, the underlying data actually speaks about concrete objects (called instances) and their relationships. In the following, we provide an overview of the types of data that can be found on the Semantic Web. Throughout the paper, it is referred to as semantic data. In fact, the term semantic data is frequently used in the Semantic Web community to refer to data that has been captured using the Semantic Web languages RDF and OWL.

RDF data is a form of semantic data that can be found most frequently on the Web. Since general data graphs can be captured using RDF, this language is sufficiently general to accommodate different kinds of data on the Web. A large amount of legacy data (given as relational or XML data) has been converted to RDF and made available on the Web, contributing to a growing Web of data. We will discuss this development later in the part on linked data in Section 2.3.5.

On the Web, semantic data might be available as data dumps, hidden behind Web APIs, embedded in Web pages or can be downloaded directly as files. It is important to note that while a schema might be available that describes the structure and semantics of the underlying data, semantic data frequently comes schemaless, especially in the case of RDF data embedded in Web pages. This means that there is no schema that is associated with the data or when available, is incomplete so that it is not possible to rely on the schema alone. Thus, we shall distinguish between fully-structured semantic data that has a schema and semi-structured se-
mantic data that does not outputs whether resource in $\mathcal{R}$ is a result to a query in $\mathcal{Q}$ or only partially.

### 2.3.4. Ontologies

The files containing semantic data that can be downloaded from the Web often represent ontologies. In the Semantic Web community, ontologies have received widespread acceptance. The notion of ontologies employed by this community is very general. Ontologies constitute rather a family of models, which might differ in the degree of expressiveness and formality, ranging from simple taxonomies and shallow conceptual data models represented in RDF(S) to expressive formal models represented in Description Logics such as OWL [SS04]. Commonly, ontologies comprise not only the conceptual part but also instances. They are more similar to databases (and knowledge bases) where the conceptual part is in fact the conceptual data model also referred to as the schema (terminological knowledge) and instances correspond to the actual data of the database (assertional knowledge of the knowledge base). Instances in fact make up the semantic data that is contained in an ontology.

### 2.3.5. Linked Data

Linked data found on the Web are basically semantic data published in RDF. The concept of linked data is about publishing and establishing links between data from different sources. The community specifically focused on this topic has worked out best practices for exposing, sharing, and connecting data on the Semantic Web. The four principles of linked data published by Tim Berners-Lee are:

- Use Uniform Resource Identifiers (URI) to identify things.
- Use HTTP URIs so that these things can be referred to and looked up (“dereference”) by people and user agents.
- Provide useful information (i.e., a structured description) about the thing when its URI is dereferenced.
- Include links to other, related URIs in the exposed data to improve discovery of other related information on the Web.

Based on these simple principles, a large amount of legacy data has been transformed and exposed as linked data. This is supported by the W3C Linking Open Data community project. Its main goal is to augment the Web with open and interlinked RDF datasets. In October 2007,
datasets consisted of over two billion RDF triples, which were interlinked by over two million RDF links. By May 2009 this had grown to 4.2 billion RDF triples, interlinked by around 142 million RDF links. Some datasets that can be found on the linked data Web are:

- **DBpedia** represents the structured data counterpart of Wikipedia. It contains data extracted from Wikipedia texts. There are about 2.18 million resources described by 218 million triples, including abstracts in 11 different languages.
- **DBLP Bibliography** provides bibliographic information about publications; it contains about 800,000 articles, 400,000 authors, and approx. 15 million triples.
- **GeoNames** provides RDF descriptions of more than 6,500,000 locations worldwide.
- **UMBEL** is a lightweight ontology of 20,000 classes and their relationships derived from OpenCyc. It has links to 1.5 million named entities from DBpedia and YAGO, another dataset that has been built to capture the information in Wikipedia.
Fig. 2.2 shows a sample of sources and data that have been made available as linked data. There are domain-independent LOD sources such as OpenCyc, DBpedia and Freebase, and sources capturing knowledge of some particular domains such as music, health care and life science.

Different kinds of links can be established. One element in a data source can be linked via a relation to one another in a different data source. Of particular interest are sameAs links, which basically denote that two RDF resources or two classes of resources represent the same real world object(s). A sample of linked data on the Web is illustrated in Fig. 2.3.

Linked data has attracted much interests from researchers of the Semantic Web community. Increasingly, this large amount of data has become a topic also for database researchers. Representing one growing Web of data, it opens new practical problems that so far, have only been studied from a theoretical point of view by database researchers in the field of Web databases and dataspaces. It is not only a testbed for existing database concepts and techniques but also introduces new challenges that are relevant for database research.

Also, linked data has attracted interests from industry. Large companies including Google, Yahoo, Microsoft have embraced this new development and are now studying various ways to commercially exploit it. Besides for Web search and other domains of commercial applications, linked data has become a subject for politics. The governments of the US and Great Britain are now releasing a large amount of linked data and elaborating on applications to expose them to citizens.
2.3.6. Embedded Semantic Data

Annotating documents with structured data has always been a goal of researchers who study new ways to improve the document retrieval experience. Recently, a W3C Recommendation called RDFa has evolved. Basically, RDFa adds a set of attribute level extensions to XHTML for embedding RDF data within Web documents. Using this standard, RDF triples can be associated with elements of an XHTML document. Also, this W3C recommendation contains the specification for the extraction of these RDF triples.

In other words, RDFa allows to talk about documents and its constituents, i.e., to produce metadata that refers to elements of a document representation. Common RDFa attributes being used to produce document metadata are:

- about and src can be used to assert the resource the metadata is about.
- property is for specifying a property for the content of an element.
- content is an optional attribute that overrides the content of an element when using the property attribute.
- typeof is an optional attribute that specifies the RDF type(s) of the resource that the metadata is about.

In the following, we show an example of using RDFa to associate metadata to the document My Book. It adds structured data to represent the title, the creator and the date of creation.

```xml
<p xmlns:dc="http://purl.org/dc/elements/1.1/"
    about="http://www.example.com/books/MyBook">
    In this book
    <cite property="dc:title">My Book</cite>,
    <span property="dc:creator">Tran Duc Thanh</span> elaborates on search on the Semantic Web.
    In particular, this book is about the retrieval of ontologies, linked data and embedded semantic data.
    <span property="dc:date" content="2010-04-02">
        April 2010
    </span>.
</p>
```

2.4. Queries on the Semantic Web

In this section, we aim to provide an overview of some common formalisms used for querying data on the Semantic Web. We discuss key-
word queries, conjunctive queries and SPARQL queries.

2.4.1. Keyword Queries

Keyword queries are basically simple lists of terms. So far, this type of queries constitutes the principle mean for searching Web resources. Most of the major Web search engines available today rely on this query paradigm.

In general, this simple formalism is valuable whenever it is difficult to express formal structured queries. Specifying structured queries requires the user to know the syntax and semantics of the query language. Further, knowledge about the schema and the underlying data is needed. These strong requirements make formalisms based on structured queries less attractive and usable for lay users – such as the typical Web users. For this reason, keyword queries have become not only popular in the IR community but also in the database and Semantic Web community. They have been employed for both document retrieval and data retrieval.

For document retrieval where the underlying resources are represented as textual data, it is natural to have queries also available as text. However, it is difficult to exploit a rich model of the information need when the underlying resource model is flat, i.e., represented as simple “bag of words”. That is, there is no much gain when structure is available in the query but not in the resource representation.

However, keyword queries also occupy a large community of database researchers. When dealing with structured data, it is natural to assume that a structured query language is more effective. It is more powerful, allowing the user to express richer models of the information needs. The structure constraints represented by these queries can be matched against structured data to obtain precise results. However, there are scenarios where users do not know the schema and the data. This is always the case when the underlying data is evolving fast. In such cases, keyword search is considered by database researchers as a principle mean to retrieve structured data. This scenario is most likely on the Semantic Web and thus, keyword search can be assumed to continue playing a substantial role in Semantic Web Search.

2.4.2. Conjunctive Queries

Conjunctive queries constitute a common fragment of widely structured query languages. They represent a restricted form of first-order queries
that match a large part of queries issued on relational databases and RDF stores. This class of queries has a number of desirable computational properties – as opposed to more powerful classes of queries such as relational algebra queries.

Basically, a conjunctive query is a first-order logic statement given by the set of formulae that can be constructed from atomic formulae using conjunction and existential quantification – but not using disjunction, negation or universal quantification. It is usually given in prenex normal form, i.e., $(x_1, \ldots, x_k). \exists x_{k+1}, \ldots x_m. A_1 \land \ldots \land A_r$, with the free variables $x_1, \ldots, x_k$ being called distinguished variables, and the bound variables $x_{k+1}, \ldots, x_m$ being called undistinguished variables and $A_1, \ldots, A_r$ are atomic formulae.

Note that conjunctive queries correspond to select-project-join queries in relational algebra and to select-from-where queries in SQL in which the where-condition uses exclusively conjunctions of atomic equality conditions. Also, as discussed in the following, it can be used to express the basic graph patterns in SPARQL queries.

2.4.3. SPARQL Queries

SPARQL is based around the notion of graph pattern matching. It provides primitives to form complex graph patterns by combining smaller patterns in various ways using conjunctions and disjunctions. The most basic pattern is a triple pattern. For instance, the pattern `$x$ name Rudi asks for all resources with the name Rudi. Combining triple patterns via conjunction yields basic graph patterns. A basic graph pattern is a set of triple patterns forming a graph, and answers to such queries must satisfy all constituents triple patterns. Other more complex patterns include group graph pattern, optional graph pattern, alternative graph pattern, and patterns on named graphs where a named graph is basically a mean to reify a set of triples.

We now show a SPARQL example for the question “Who works at KIT?”:

```sparql
PREFIX abc: <http://example.com/exampleOntology#>
SELECT ?researchers
WHERE {
  ?researchers abc:work ?institute ;
  ?institute abc:name 'KIT'.
}
```
Chapter 2. Semantic Web Search

This query represents a basic graph pattern consisting of two triple patterns. Clearly, this kind of patterns can be represented using conjunctive queries where every triple pattern corresponds to an atomic formulae.

### 2.5. Semantic Search

In this section, we introduce our model for Semantic Search and discuss its extension that constitutes the primary subject of this book, namely the model of Process-oriented Semantic Search.

#### 2.5.1. Basic Semantic Search Model

We firstly define a basic Semantic Search model where the semantics of resources and queries are explicitly represented and exploited for search.

**Definition 5 (Basic Semantic Search Model)** The Basic Semantic Search model is a quadruple \( \langle R, Q, S, M(Q, R, S) \rangle \) where

1. \( R \) is the resource model, which is a set of syntactic representations for the underlying resources.
2. \( Q \) is the query model, which is a set of syntactic elements representing the user information needs.
3. \( S \) is the semantic model capturing the meaning of the represented information needs and resources.
4. \( M(Q, R, S) \) is the semantic matching framework which models the relationship between resource representation and query representation. It specifies the notion of matching, which incorporates the associated semantic model to compute whether the resource representation contains an answer to the query representation. In particular, there is a matching function \( M : Q \times R \times S \mapsto [0, 1] \), which for a query representation \( q \in Q \), a resource representation in \( r \in R \) as well as the associated semantic model \( S \), outputs the degree to which \( r \) is a result to \( q \).

This is a general model of Semantic Search. Just like the general search model presented previously in Section 2, this one does not dictate whether the subjects of interest are data or documents. The resources could be structured data or some representations of documents. Also, it does not specify concrete requirements for the matching framework, which could be instantiated by any kinds of data retrieval or document
retrieval techniques. The only difference to the general search model is the use of a semantic model. We will now elaborate on different types of approaches which fall into this category of Semantic Search.

2.5.1.1. The Database Perspective

In retrieving data, semantics has always play an essential role in database (DB) systems. The foundation of database technologies manifested in concepts such as the relational model that provide formal semantics to both resources and queries, i.e., the model behind relational data and relational query (an important part of SQL) is in fact a fragment of first-order logic. This semantics is taken into account by database query engines to compute answers that precisely match the semantics of queries. More precisely, these engines perform binary matching that incorporates the structure and semantics into account. A resource is either an answer or not. Thus, strictly speaking, database approaches do not completely fit the Semantic Search model presented above because it does not consider the aspect of ranking. Yet, it is important to point out that they do employ a well-defined semantic model.

Using this mechanism, information needs of different complexity have been served. In different commercial systems, we find the application of DB technologies in the form of simple product search (e.g. in commercial shopping sites like Ebay and Amazon) up to complex relational queries that are used to populate complex dynamic Web pages.

While the relational model is the one most widely used in practice, there exist also domain-specific solutions that are built upon more expressive semantic models. In the realm of deductive databases for instance, semantics represented in data models such as Datalog or F-Logic ontologies is used to perform reasoning, with the goal of understanding and delivering not only explicitly asserted information, but also inferred one. Further down this steam, semantic models even more expressive are used by knowledge-based expert systems for answering complex questions.

2.5.1.2. The Information Retrieval Perspective

In the IR community, the representation of the information needs and resources is achieved primarily using lightweight models that operate at the level of words. In particular, the predominant paradigm is based on keyword-based representation of queries and bag-of-words representation of resources. This works and most importantly, scales well for topical search, i.e., to retrieve documents based on topics expressed in terms of
keywords, but is not sufficient to address the kind of complex information needs that can be served by database and knowledge-based systems.

The need for going beyond the matching purely based on words and statistical dependencies that exist between them has been clearly expressed by leading researchers in the field. In the early years of IR research, Rijsbergen clearly argued for the use of semantics [vR86]. In order to return precise results to the user, it is essential to compute whether the resources’ content semantically entail what the user asks for [vR86]. Quoting a paper from W.B. Croft et al. published more than twenty years ago [Cro86]: “The statistical approach has many advantages and can achieve a reasonable level of effectiveness with techniques that are very efficient. However, it appears that to achieve significant improvements in retrieval effectiveness compared to current techniques, systems must be designed to acquire and use explicit domain knowledge”. This explicit domain knowledge has been captured using semantic models of varying degrees of expressiveness. Semantic networks have been used to augment the models based on terms [Sho81]. This provides a richer representation, which is exploited for assisted query formulation, query expansion, as well as for relevance propagation in ranking. Also, linguistic models particularly Wordnet [Voo93] and the Rogets thesaurus [Bra93] have been widely used.

Recently, the IR community continues to embrace the vast body of explicit knowledge available on the Semantic Web. There are approaches which not only use semantic models to enrich terms, but in fact, employ both the conceptual and the data part. In particular, there are systems that use ontologies including the semantic data contained in them [CFV07].

2.5.1.3. The Semantic Web Perspective

Semantic Search from the this point of view is more similar to the database perspective in the sense that the main goal is to obtain precise answers for complex questions. That is, as opposed to the IR perspective that is centered on document retrieval, the database and the Semantic Web perspectives are focused on data retrieval. However, while database and knowledge-based systems are adopted to solve specific problems of a particular domain, semantics is taken by these communities to the wider context of the Web. Semantic models used here are ontologies available in RDF and OWL. On the Semantic Web, there is more data that is more heterogeneous, i.e., data coming from different sources that differs in syntax, formats and quality. Thus, unlike traditional database systems, Semantic
Web Search engines are built with emphasis on the ability to scale in terms of both volume and the number of sources, to be domain independent and to perform ranking.

To achieve this, they are built upon concepts and technologies from database and knowledge-based systems. General concepts for data management (e.g. indexing and data partitioning) as well as specific concepts for managing graph-structured data and graph databases (e.g. structure indexes from XML databases) have been adopted to deal with ontologies and semantic data available in RDF and OWL. For dealing with more expressive semantic models, techniques commonly applied in deductive databases (e.g. magic set optimizations) have been adopted to improve the efficiency and scalability of reasoning. However, to deal with scale and to perform ranking, techniques that are used by the IR community have also been adopted. For instance, the inverted index has been used to scale to a large amount of data and to support term-based matching. Statistical dependencies of words such TF-IDF as well as authority-based metrics such as PageRank that is popular in IR Web search engines, have been used for ranking. We will discuss this in more details in the part on the state of the art of Semantic Web Search in Chapter 3.

To summarize, semantics has proven to be useful for search in many aspects. There are commercial exploitations, which used semantics in different ways. Thus, the notion of Semantic Search is thus very broad. Semantic Search covers the wide range of systems employing and exploiting semantic models of varying expressiveness as well as a large body of diverse concepts and techniques developed by different communities.

### 2.5.2. Process-Oriented Semantic Search Model

The Basic Semantic Search model and the approaches that instantiate it focus on the technical aspect of matching the query to resources. Cognitive approaches to IR problems have shown that the retrieval effectiveness can be significantly improved when the user is taken into account [Ing94, SO98]. In particular, improvement can be achieved by considering the entire retrieval process instead of the matching step only. In Fig. 2.4, we illustrate the models and steps involved in a process-oriented view of Semantic Search. Just like the basic model, the Process-oriented Semantic Search model is centered on the use of semantics. However, it goes beyond that to recognize that Semantic Search also targets complex information needs. To take the resulting complexity facing the users into
account, semantics should be exploited also for helping users in constructing queries and in handling and understanding results. We now provide the definition of this process-oriented model:

![Diagram of the process-oriented search model](image)

**Figure 2.4.** Process-oriented search model.

**Definition 6 (Process-oriented Semantic Search Model)** The Process-oriented Semantic Search model is a tuple \( \langle R_S, R_P, Q_U, Q_S, Q_P, S, M(Q_S, R_S, S), T(Q_U, Q_S, S), P_R(R_S, R_P), P_Q(Q_S, Q_P) \rangle \) where

1. \( R_S \) is the system-resource model, which is a set of syntactic elements constituting the internal representation of the system’s resources.
2. \( R_P \) is the presentation-resource model, which is a set of presentation elements constituting the external representation of the system’s resources.
3. \( Q_U \) is the user-query model, which is a set of elements representing the information needs as provided by the user.
4. \( Q_S \) is the system-query model, which is a set of elements constituting the internal representation of the user information need employed by the system.
5. \( Q_P \) is the presentation-query model, which is a set of elements constituting the external representation of the information need provided by the system.
6. Just like in the basic model, \( S \) is the semantic model capturing the meaning of query and resources. \( M(Q_S, R_S, S) \) is the semantic
matching framework that models the relationship between resource representation \( R_S \) and query representation \( Q_S \) and specifically, specifies the notion of matching under consideration of the semantic model \( S \).

7. \( T(Q_U, Q_S, S) \) is the semantic translation framework which models the relationship between user-query representation \( Q_U \) and system-query representation \( Q_S \). In particular, there is a translation function \( T : Q_U \times Q_S \times S \mapsto [0, 1] \), which for the given query \( q_u \in Q_U \) provided by the user, an internal representation in \( q_s \in Q_S \) employed by the system, as well as the associated semantic model \( S \), outputs the degree to which \( q_s \) is a translation of \( q_u \).

8. \( \mathcal{P}_R, \mathcal{P}_Q \) are the semantic presentation frameworks which model the relationship between internal representation and external representation of queries and resources, i.e., \( Q_S, Q_P \) and \( R_S, R_P \). Under consideration of the semantic model \( S \), it specifies the mapping of syntactic elements in \( R_S, Q_S \) to presentation elements in \( R_P, Q_P \).

This general notion of Process-oriented Semantic Search recognizes the gaps between internal representation of the resources and queries and what can be best specified, understood and handled by the user. For supporting complex information needs, an expressive formal query language might be used for \( Q_S \). Specifying such queries is a complex task. Also, the system output might be not only a single answer or a single resource description but also, complex results encompassing sets of resources and their relations. The frameworks for translation and presentation are introduced to bridge these gaps.

Within these frameworks, there are no constraints as to what is the query provided by the user and how the queries and resources are presented to the user. In particular, the representations and presentations of resources and queries might be the same for both user and system, i.e., \( Q_U = Q_S = Q_P \) and \( R_S = R_P \). In this case, query translation and presentation are not involved and process-oriented search amounts to basic search that is based on matching only.

However, for supporting complex queries and results, the process-oriented model goes beyond the basic one to enable different ways of specifying queries, and of presenting results. As examples, user queries \( Q_U \) might be specified using predefined forms, keywords or natural language. Presentation components used for \( Q_P, R_P \) are not restricted to
syntactic elements but might include graphical elements delivered to the user in the form of rich widgets. The methods for implementing the translation framework $T$ and the presentation frameworks $P_R, P_Q$ shall exploit the underlying semantics to map the more assessable models targeted at users to the models internally used by the system, and vice versa.

2.6. Semantic Web Search

As this point, we have established the notion of semantics, discussed different kinds of semantic models and arrived at the definition of Semantic Search and its extension Process-oriented Semantic Search. Now, we would like to discuss the subject of this book, namely the concept of Semantic Web Search. We begin with a working definition of Semantic Web Search:

**Definition 7 (Semantic Web Search Model)** The Semantic Web Search model is a tuple $\langle R, Q, S, M(Q, R, S) \rangle$ where

1. the resource model $R$ is made up of semantic data available in RDF or OWL, and which comes as ontologies, linked data or embedded semantic data.
2. This data might be associated with a semantic model $S$.
3. The query model $Q$ is of the type keyword query, conjunctive query or SPARQL query.
4. $M(Q, R, S)$ is the semantic matching framework defined by the function $M : Q \times R \times S \mapsto [0, 1]$, which for a given query representation $q \in Q$, a resource representation $r \in R$ as well as the associated semantic model $S$, outputs the degree to which $r$ is a result of $q$.

Clearly, this model is a refinement of the Semantic Search model. The difference is that it explicitly deals with the types of data and queries that can be found on the Semantic Web. Also, while it does not rule out document retrieval, it is rather focused on the retrieval of semantic data. Document retrieval is only supported when the documents of a given collection are represented as semantic data – using the ontology-based document model presented in Section 2.3.2 for instance. Clearly, results that can be obtained when processing queries against data on the Semantic Web might vary in quality, recency, relevance etc. Thus, the aspect of result ranking is incorporated into this working definition.
Another important aspect is the use of semantic models. On the Semantic Web, a semantic model might exist, but in many cases, is actually incomplete or simply not available. A semantic model shall be used for retrieving semantic data but in cases it does not exist, is is necessary to depart from this assumption and to reach for alternative solutions.

In this book, we present a Semantic Web Search solution that relies on the automatic computation of lightweight semantic models from data. Thus, it also uses a semantic model and can indeed be considered as a Semantic Search solution. In particular, we take the process-oriented perspective on Semantic Search as discussed in the previous section. This book presents a \textit{Process-oriented Semantic Web Search} approach.

\subsection*{2.7. Conclusions}

We have introduced the main search tasks, namely the ones of data and document retrieval. We have defined the concrete problem tackled by this book, i.e., the Semantic Web Search problem, which is about retrieving data on the Semantic Web. Clearly, this is primarily a data retrieval problem. However, we have also discussed techniques we proposed for modeling and managing metadata. They allow documents to be modeled in terms of semantic data in RDF or OWL. Using such expressive models of documents, document retrieval can also be treated as a data retrieval problem.

For retrieval, keyword queries, conjunctive queries and SPARQL queries have been recognized as the main and most frequently used paradigms on the Semantic Web. We have discussed the different types of data that can be found on the Semantic Web, from ontologies to linked data to embedded semantic data. For solving the problem of Semantic Web Search, we might be able to rely on a semantic model. On the Semantic Web, this is not always the case such that a semantic model might have to be computed from the data.

Approaches that use a semantic model for search are called Semantic Search. Semantic Web Search and especially the approach we deal with in this book constitute one particular type of Semantic Search. We have argued for an extension of this Semantic Search notion to accommodate the complexity the user faces when dealing with complex information need. We have presented a Process-oriented Semantic Search model that includes additional support for query construction and result presentation.
Chapter 3

The State of the Art of Semantic Web Search

3.1. Introduction

Different approaches and implementations have been proposed for searching Semantic Web resources. There exists a wide range of systems, which can be used for this purpose. The variety of systems range from reasoning engines such as Pellet\(^1\) and Racer\(^2\) that can be used to infer and retrieve knowledge from expressive ontologies, RDF Stores such as Sesame\(^3\), Jena\(^4\) and Virtuoso\(^5\) that focus on managing and querying data and search engines specifically built for retrieving Semantic Web resources such as Watson, Swoogle, SWSE, FalconS and Hermes that support retrieval on the large scale albeit using more simple types of queries.

To collate these various approaches and to better understand what the concept of Semantic Web Search entails, this chapter discusses the state of the art of techniques used by existing Semantic Web Search engines. Fig. 3.1 illustrates the topics and the structure of this chapter.

In Section 3.2, this chapter starts with a discussion on the objectives and challenges of Semantic Web Search. Section 3.3 discusses the different techniques that have been proposed to crawl semantic data coming from different sites and repositories. The techniques for storing, indexing

\(^1\)http://clarkparsia.com/pellet  
\(^2\)http://www.sts.tu-harburg.de/~r.f.moeller/racer/  
\(^3\)http://www.openrdf.org/  
\(^4\)http://jena.sourceforge.net/  
\(^5\)http://www.openlinksw.com/
and managing data as well as the ones used for making it available for search and for processing user queries are presented in Section 3.4.

In the Semantic Web setting where the amount of resources is large, a query might be associated with a large number of candidate matches. Typically, the user is however interested only in a few results that are most relevant. Performance is one another consideration. Returning all results that are possibly relevant for the given query might be not affordable. Section 3.5 introduces ranking techniques which address these issues. They have been proposed to apply an upper bound on the number of results to be returned and in particular, to return only the most relevant results.

While the techniques discussed in the previous sections are rather independent of the search scenario, the application of search on the Web introduces specific challenges. Section 3.6 presents approaches specifically designed for the multi-source scenario, the challenges related to Semantic Web Search they facing, and the specific techniques they employ to master them. In particular, the problems of federated query processing and data integration are discussed in detail.

This chapter concludes in Section 3.7.

3.2. Objectives and Challenges

The explosion of resources available can be witnessed at different levels, from the personal sphere to the enterprise setting up to the realm of the Web. End users collect, produce and share resources with other people in their social networks or publish them on the open Web. At the level of enterprises, the volume of data to be managed is often in the order of Terrabytes. This refers to structured data residing in relational or XML databases only, while the amount of textual data in the form of documents, as well as images, videos, audios and multimedia documents stored in separate repositories might be considerably larger. On the Web, there are millions of documents and an equally large amount of structured data kept behind APIs and metadata associated with Web documents and services. As we have motivated in the introduction of this book, there is also a trend suggesting that increasingly, these different types of data are made publicly available as semantic data in the form of RDF.

As a result, semantic data is not only increasing in volume, but also in terms of heterogeneity and complexity. Here, heterogeneity refers to the diversity of sources the data comes from as well as the types of data. There are textual data in documents, semi-structured as well as fully-
structured semantic data available in RDF and OWL. These different types of data do not come in isolation but might be intertwined. For instance, RDF resources are often associated with a large amount of text, e.g. the description of a RDF person is often given as long text. Vice versa, RDF data might be embedded in text documents. Complexity in this sense includes both the structure and the aspect of semantics. Efforts invested in linking data across different sources on the Web has resulted in data becoming more integrated, capturing complex interconnected structures that might range over different domains. Further, data might come with complex schemas capturing rich semantics, based on which reasoning can be performed to infer additional knowledge.

Clearly, this development creates opportunities as well as challenges. Recognizing the chances that arise from this increasing wealth of resources on the Web, as well as the economic value of the associated business, new search engines providers such as Hakia, Powerset and Cuil have entered the market to compete against Google for market share. Likewise, research interest is stirring up. A large body of work has been proposed by
researchers from different communities, including database, IR and Semantic Web. Behind these commercial and scientific endeavors lies one central theme: the use of semantics for improving the search experience. In the previous section, we have studied this line of work under one a single model, the Semantic Search model. We are particularly interested in the type of Semantic Search that is concerned with the retrieval of semantic data, which we coined Semantic Web Search. The overall objectives targeted by Semantic Web Search approaches can be summarized as follows:

- **More Results with High Degree of Relevance (Leveraging Data Volume)** With the increasing availability of resources, there are also more candidate results, which might be relevant for the user needs. Further, more sophisticated notion of relevance can be taken into account. Resources can be selected to match not only the content and structure constraints of the query but also other dimensions of the user needs such as preferred media types (e.g. text vs. audio) or content format (e.g. full version vs. snippets).

- **More Integrated Results (Leveraging Data Heterogeneity)** Besides relevance, the notion of result also becomes more sophisticated. A result might be a lengthy resource descriptions or summarized compact version of them that are computed at query processing time. Moreover, it might be an integrated unit of content that combines data of different types and formats coming from different sources.

- **Addressing more Complex Needs (Leveraging Data Complexity)** Semantic data, especially the one associated with rich schemas, allows for addressing more complex needs. The structure and semantics in the data can be exploited to answer queries that go beyond simple list of keywords. Instead of content-based matching only, structural matching technique up to logical reasoning that takes the semantics captured in the schema into account, might be used for search. The results of queries will be more precise, matching not only the content but also the structure and semantics of the query.

Harnessing these benefits is important for the productivity of the individual users, and might result in enormous economic benefits for search providers as well. The main challenges to this end are:

- **Data Management** For searching, data has to be crawled, indexed and updates have to be made to reflect changes in the original data.
Accomplishing these tasks in an affordable manner and in particular, scaling them is difficult, given an ever increasing amount of data that has to be managed. While it has been shown by major search engine providers that textual data in documents can be managed at Web-scale, it remains a research question how to scale databases beyond the realm of Terrabytes of structured data. In particular, managing a large amount of semantic data is still an open issue.

- **Integration** In order to exploit heterogeneous data, links have to be established to connect structures that exist across different sources. A larger number of links allows more complex needs to be addressed, and more integrated results to be delivered. There exists a large body of work on integration, which for specialized domains, has achieved precision in the order of 90 percent. However, these techniques do not scale to a large amount of data that possibly, covers different domains and is available in different formats and types. Not only the quality degrades as the data volume increases but also the efficiency. Existing integration methods, when applicable at all, require enormous amount of time on large amounts of data and thus, are not affordable in large-scale settings.

- **Querying** While the structure and semantics available in the data can help to address more complex tasks, scalability is a serious problem in practice. Matching complex structures as supported by databases does not scale beyond the realm of enterprises. For more complex data analysis tasks, specific solutions in the form of data warehouses are employed, which rely on complex index structures that have to be created and maintained separately from the data. Even more complex and thus worst in tractability and scalability is the use of semantics for reasoning. Yet, reasoners have be applied only in very specialized domains where the setting conditions can be controlled and real-time responses are not critical. In short, for querying, for data analysis and in particular for reasoning, scalability is a serious problem when moving towards a large-scale Web scenario.

- **Ranking** This is essential for focusing on the relevant ones out of the large amount of resources. Both the increase in volumes and heterogeneity pose challenges for ranking. There must be mechanisms in place to filter resources that are out of date, low quality or generally speaking, are not relevant to the user need. That is, there must be mechanisms that support a more sophisticated notion of
relevance that goes beyond factors derived from language and text and content-based matching to consider the structure as well as the semantics of the queries and the underlying resources.

In this chapter, we aim to recapitulate the body of work on Semantic Search by elaborating on the state of the art. We focus on the retrieval of the types of data that can be found on the Semantic Web. Thus, it is rather a study of Semantic Web Search than Semantic Search. Fig. 3.2 shows an overview of the main tasks that are essential for Semantic Web Search. These tasks are crawling, storing, indexing, querying, ranking and dealing with search in a multi-source scenario. To deal with data from multiple sources, it is necessary to perform data integration and federated query processing. We will now elaborate on the concepts and techniques that have been proposed to perform these tasks.

3.3. Crawling Semantic Data

The starting point of a Semantic Web search system is obtaining semantic data. For harvesting data on the Semantic Web, different crawlers have been developed which conceptually, are based on the crawling technologies widely used by traditional Web search engines. The main difference is that instead of standard Web resources available in HTML, XML and other content formats, Semantic Web crawlers are specifically focused
Chapter 3. The State of the Art of Semantic Web Search

on semantic data. As discussed, this type of data might be kept in RDF and OWL ontologies, available as data dumps or embedded in Web documents.

For identifying the sources of semantic data, Semantic Web crawlers employ different strategies. The most straightforward and effective technique is to rely on existing and well-known repositories, like Swoogle or the Protégé ontology library. Specialized crawlers for these repositories fetch semantic data by sending queries that have high coverage. These are queries that are likely to cover a large number of ontologies and semantic data documents contained in the repositories. Other crawlers heuristically explore Web pages to discover new repositories and to locate OWL and RDF documents. For instance, they send queries to Google containing the keywords “filetype:owl”. The goal of crawling is to identify and collect valid semantic data. For validation, crawlers check both the document type and format as well as the actual content. This can be supported by various ontology management softwares that include parsers for validating the syntax as well as specific tools for validating semantic consistencies. Any document that can not be parsed or when its content is semantically inconsistent might be omitted.

Already collected semantic documents are frequently re-crawled to discover and incorporate changes in the current content or to add new data that has become available in the mean time. Also, the collected data is usually inspected for links to other locations of semantic documents. In addition to standard hyperlinks used in Web pages, there are several relations between ontologies and between semantic data elements that can be followed. There are either declared ones such as owl:import and rdfs:seeAlso or implicit ones such dereferencable URIs and used vocabularies.

Similar to traditional Web crawling, collecting semantic data involves ethical issues. Crawlers should behave in accordance with accepted ethics of good behavior. The aspects of privacy and crawling costs related to a good behavior are partially addressed by the robots exclusion protocol. This protocol enables site owners to partially disallow crawling on their site. Following this protocol is a general practice accepted and implemented by most Semantic Web crawlers. We will now discuss some crawlers we found in literature:

- **Swoogle [DFJ⁺04](http://swoogle.umbc.edu/)** is the first search engine indexing Semantic Web documents, currently including those written in RDF/XML and other RDF serialization formats
such as N3. Also, it contains documents embedding RDF/XML fragments. The crawler used by Swoogle adopts a hybrid approach to harvest the Semantic Web. In particular, it combines the manual identification of repositories with heuristic Google-based crawling. Swoogle provides a service for (1) checking if a Uniform Resource Locator (URL) has been indexed (2) tracking the previous versions of the Semantic Web document retrieved from a particular URL and (3) for submitting new URLs that should be employed by Swoogle for crawling.

- **Watson** [SBG+06] (http://watson.kmi.open.ac.uk/) collects online available semantic data primarily by exploring links available in a seed collection of semantic data documents. Compared with standard Web crawlers, Watson is not dealing only with Web pages but also with semantic data. So, it employs additional tools to validate the format and content of the data discovered during the crawling process.

- **Falcon**s (http://iws.seu.edu.cn/services/falcons/) also employs a collection of seed URIs, which are obtained in three ways. Firstly, a list of phrases are extracted from category names of the top three levels of the Open Directory Project (dmoz.org). These phrases are sent to Swoogle and Google to obtain URIs. Secondly, URIs of RDF documents found in online repositories such as pingthesemanticweb.com and schemaweb.info are incorporated into the seed collection. Thirdly, URIs from datasets published in the LOD project were manually added. This collection of URIs are then processed by a multi-threaded crawler. It dereferences the URIs, performs content negotiation and downloads the semantic data documents.

- **MultiCrawler** [HUD06] has a five steps pipelined architecture to crawl and index data from the Semantic Web. These steps are fetch, detect, transform, index, and extract. First, the data is fetched from the Web. Then, the type of content, e.g. RDF, WSDL, GIF etc., is detected. During the transform phase, the data is converted to a common data format. An index is built, which is finally used in the extract phase to query for URIs of additional sources.
3.4. Managing and Querying Semantic Data

Existing work on managing and querying Semantic Web data can be roughly divided into three categories: (1) IR-based approaches, (2) DB-based approaches and (3) native approaches. IR-based approaches leverage concepts and techniques from IR whereas DB-based approaches are built upon the results of database research. Native approaches are concerned with solutions that are tailored and specifically optimized for semantic data.

3.4.1. IR-based Approaches

Prominent examples of this category include the various Semantic Web Search systems such as Falcons [CQ09], Sindice [TDO07], Swoogle [DFJ+04] and Watson [SBG+06]. These systems crawl Semantic Web documents and use inverted indexes managed by an underlying IR engine to provide lookup functionalities. Keywords submitted by the user are matched against the indexed resources. Then, the relevant ones are ranked according to the matching scores returned by the underlying IR engine. These systems take a pure IR approach. In particular, they support the processing of content-based constraints expressed in terms of keyword queries but lack support for structured queries. To address this shortcoming, we have proposed Semplore [WLP+09]. It is built upon scalable IR technologies to support simple semantic data lookup, similar to existing Semantic Web Search systems. This feature is however extended to support also structured queries. We will now discuss this solution in more details.

IR indexing is based on the concepts of documents, fields (e.g., title, abstract), and terms. Using inverted indexes, IR engines can efficiently retrieve documents for a given query consisting of a boolean combination of \((\text{field}, \text{term})\) pairs. Current Web search engines have proven that this technique scales to the large quantity of documents on the Web.

Intuitively speaking, if we treat RDF resources as documents and their associated classes as terms, we can retrieve all individuals of a given class by inputting the class name as a query term. Extending this intuition, we can answer many types of semantic queries using IR engines, when we transform semantic data to documents, fields, and terms in a specific way, as shown in Table 3.1. After the transformation, the semantic data is stored in inverted indexes and retrieved using an IR engine. In particular, the
retrieval of RDF resources based on `subClassOf`, `superClassOf`, `subRelOf`, `superRelOf`, `type`, and `text` are supported.

<table>
<thead>
<tr>
<th>Document</th>
<th>Field</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*Class C</td>
<td><code>subClassOf</code></td>
<td>labels of sub-classes of C</td>
</tr>
<tr>
<td></td>
<td><code>superClassOf</code></td>
<td>labels super-classes of C</td>
</tr>
<tr>
<td></td>
<td>text</td>
<td>tokens in textual properties of C</td>
</tr>
<tr>
<td>3*Relation R</td>
<td><code>subRelOf</code></td>
<td>labels of sub-relations of R</td>
</tr>
<tr>
<td></td>
<td><code>superRelOf</code></td>
<td>labels of super-relations of R</td>
</tr>
<tr>
<td></td>
<td>text</td>
<td>tokens in textual properties of R</td>
</tr>
<tr>
<td>4*Resource i</td>
<td>type</td>
<td>labels of classes i belongs to</td>
</tr>
<tr>
<td></td>
<td><code>subjOf</code></td>
<td>all relations <code>(i, R, ?)</code></td>
</tr>
<tr>
<td></td>
<td><code>objOf</code></td>
<td>all relations <code>(? , R, i)</code></td>
</tr>
<tr>
<td></td>
<td>text</td>
<td>tokens in textual properties of i</td>
</tr>
</tbody>
</table>

Table 3.1. Transforming semantic data to documents.

Besides these lookup functionalities, we proposed PosIdx, a position-based index to index, retrieve, and to join relation triples for supporting structured queries. In an inverted index, each term is associated with a posting list of documents containing it. In addition, for each of these documents, there is a list of positions (stored in a position list) indicating where the term appears in it. In PosIdx, relation names are indexed as terms and the subjects are stored as documents. The objects of a relation are stored in the position list. In other words, given the triple `(s, R, o)`, the object `o` is stored in the position list of the term `R` in the document `s`. An example index structure is depicted in Fig. 3.2, where `person2` and `person3` are objects stored in the position list of subject `film1` for relation `directedBy`. We treat `directedBy` as a term, `film1` as a document, and `person2` and `person3` as the positions `directedBy` appears in the document. The index structure is symmetric since the objects of a relation represent the subjects of the inverse relation. That is, `subjOf` is treated as a field used for indexing instances of a relation `R` and `objOf` is used for indexing instances of the relation `R−`.

The physical storage of the inverted index and the retrieval of documents on top of it have been thoroughly studied in the IR community. Many optimized methods have been developed to improve the efficiency of index management, such as byte-aligned index compression [SWYZ02] and self-indexing [MZ96]. These optimizations can be leveraged for the efficient retrieval of relations. Furthermore, in the proposed PosIdx method, relation objects enjoy the benefit of spatial locality for fast access, since
positions of a term are usually physically placed together, i.e., are stored contiguously in the inverted index.

Based on the proposed index, Semplore reuses the IR engine’s merge-sort-based boolean query evaluation method and extends it to answer structured queries. We will now introduce and explain some basic operations. We generalize the notion of a posting list to an Ascending Integer Stream (AIS) which can be accessed from the smallest integer to the largest one.

- **Basic-retrieval** \( b(f, t) \): Given a field \( f \) and a term \( t \), \( b(f, t) \) retrieves the corresponding posting list from the inverted index. The output of this operation is an AIS. For example, according to the structure described in Table 3.1, \( b(\text{type}, \text{ChineseActor}) \) will retrieve all resources that have \text{ChineseActor} as class.

- **Merge-sort** \( m(S_1, op, S_2) \): \( S_1 \) and \( S_2 \) are two AISs and \( op \) is a binary operator which can be \( \cap \), \( \cup \) or \( - \). Merge-sort computes \( S_1 \ op \ S_2 \) and returns a new AIS. Merge-sort can be nested to compute boolean combinations of multiple AISs. IR research has developed efficient algorithms to do nested merge-sort on AISs.

- **Mass-union** \( u(S, R) \): Given a set of subjects \( S \) and a relation \( R \), this operation returns the union of object sets \( \{ o \ | \ (s, R, o) \} \) over every subject \( s \) in \( S \), i.e., \( u(S, R) = \bigcup_{s \in S} \{ o \ | \ (s, R, o) \} \). It also sorts the union set to ensure the returned result is an AIS.

- **Relation Expansion**: To evaluate a structured query, we need an additional operation for evaluating query edges. The relation expansion operation \( \bowtie (S_1, R, S_2) \) is defined for this purpose. The input to this operation is a relation \( R \) and two AISs \( S_1 \) and \( S_2 \) that contain individual IDs for relation subjects and objects, respectively. The operation computes the set \( \{ y \ | \ \exists x : x \in S_1 \land (x, R, y) \land y \in S_2 \} \) and returns it as an AIS. For example, \( \bowtie (\land (\text{“action film”}), \text{directorBy}) \).
directedBy, $\lambda$(HongKongFilmDirector)) is used to find all Hong Kong film directors who have directed some action film. This join operation is not directly supported by current IR engines. We propose to evaluate it through a combination of the basic operations discussed above: First, we compute the subjects that have $R$ as relation, i.e., $S = m(S_1, \cap, b(subjof, R))$. Second, we find the set of objects for each subject $s \in S$, i.e., $g(s, R) = \{o \mid (s, R, o)\}$. Third, we union the object sets for these subjects and sort the result set to obtain a new AIS $S_O$ where $S_O = u(S, R) = \bigcup_{s \in S} g(s, R)$. Finally, we do a merge-sort $m(S_O, \cap, S_2)$ to obtain the final result. Fig. 3.3 illustrates these four steps to calculate $\bowtie (\lambda(“action film”), directedBy, \lambda$(HongKongFilmDirector)).

A structured query is processed by traversing the edges and evaluate them using the operations discussed above.

### 3.4.2. DB-based Approaches

IR technologies have been embraced and have become a viable alternatives for managing RDF data. Yet, most of the commercial RDF stores available for industry usage are based on database technologies. In particular, relational databases have been widely used for Semantic Web data storage. Jena and Oracle RDF store [CDES05] are examples thereof, which use a relational triple table of three columns ($s, p, o$) to store all RDF data.

---

\(6\)If the relation is $R^-$, replace subjof with objof.
RDF statements. This basic data organization scheme is also called the *triple-based organization*, where one big three-columns table is used to store all triples. To avoid the many self-joins on this possibly very huge table, *property-based partitioning* is suggested [WSKR03], where the data is stored in several “property tables”, each containing triples related to one particular entity type. For instance, this design is implemented by Jena2. Yet another scheme is employed by SOR [MWL+08], an extension to IBM DB2 that focuses on RDF data management. It uses multiple “special-purpose tables”, each of which keeps one particular of type of RDF edges. The most well-known method for data organization is undoubtedly the concept of *vertical partitioning*, which has been awarded at the VLDB conference in 2007. It has been proposed to decompose the data graph into $n$ two-columns tables, where $n$ is number of properties [AMMH07]. As opposed to property tables, this scheme can avoid the storage of null values and the corresponding problem of sparsity. In this book, we will discuss one another method in detail in Section 6.3 of Chapter 6, which we coined *structure-based partitioning*. The idea here is to physically group together data elements that are similar in structures. This way, data elements in groups with structures that correspond to a given query more likely contribute to the final results. In other words, this partitioning method can reduce IO by avoiding to load data that does not satisfy the structure constraints of the query.

Join processing can be greatly accelerated when the retrieved triples are already sorted. Through vertical partitioning, retrieved data comes in sorted fashion, enabling fast *merge joins* [AMMH07]. This makes a crucial difference because in practice, evaluating joins this way comes close to time performance linear to the size of the data. In this regard, we also contributed to the state of the art by proposing a two-level query processing procedure called *structure-aware query processing*. The second level is the data-level. At this level, data is loaded and joined to answer a particular query, just like in standard query processing. However, this processing is performed only for some parts of the query, which have not been pruned away during the first step. These parts are those containing content constraints. The first step is coined structure-level processing, which consists of join and loading operations performed on a semantic model (also called structure index) that is much smaller than the actual data. This step helps to locate the groups of data that satisfy the structure constraints of the query, and to prune away those query parts containing only structure constraints that have already been verified during the process. Section 6.3
of Chapter 6 discusses this approach in detail.

### 3.4.3. Native Approaches

Unlike the DB-based solutions, which leverage the capabilities of an underlying off-the-shelf database and extend them for the specific case of RDF data management, there exist also RDF stores that are built from scratch. In particular, stores such as Hexstore [WKB08], RDF-3X [NW08] and YARS2 [HUHD07] have been designed and specifically optimized for the management of RDF data. They feature tailored index structures and customized procedures for RDF query processing and optimization. We call the approaches behind this line of work native approaches. A popular technique used is *multiple indexing*. It is based on the creation of several indexes for the data for supporting different lookup patterns. The scheme with the widest coverage of access patterns is used in YARS [HD05], where six indexes are proposed to cover 16 possible access patterns of quads [HD05], i.e., any of the subject, predicate, object and context part of a quad pattern can be either a constant or a variable. Given a query consisting of a single pattern, query answering using this scheme amounts to one single lookup.

In [WKB08], *sextuple indexing* has been suggested, which generalizes the strategy in [HD05] such that for different access patterns, retrieved data comes in a sorted fashion. In fact, this scheme combines multiple indexing with vertical partitioning to support fast merge joins also on complex query patterns. Thus, this indexing technique goes beyond lookup operations to support more efficient join processing. Along this line, entire join paths have been materialized and indexed using suffix arrays [LH05]. A new kind of path index based on judiciously chosen “center nodes” coined GRIN has also been proposed [UPS07].

For optimal query processing, authors behind the RDF-3X system have proposed a technique for query plan optimization. They reported that further efficiency gains can be achieved by leveraging dynamic programming for finding optimal bushy plans [NW08].

In [MWL+08], it has been reported that native stores reduce load and update time, possibly due to the physical organization and indexes that are more tailored to RDF. DB-based solutions provide better query optimization. Thus, it seems there is no single system, but rather a combination of different concepts that makes up the state of the art in RDF data management and query processing. In particular, vertical partitioning [AMMH07]
is the candidate for physical data organization, IR-based technologies provide a viable alternative for indexing not only textual data but also semantic data, multiple indexing strategies [HD05, WKB08] enable fast lookup, and tailored query plans [NW08] result in fast performance for complex join processing.

3.5. Ranking Semantic Data

The ranking of results has been extensively studied in the IR community. Ranking metrics proposed so far can be divided into two broad categories: (1) query-independent ranking and (2) query-dependent ranking. The first category captures the “authority” or “importance” of a given resource regardless of the query. The latter category is used to estimate the relevance between a query and a resource. Recently, much work has been devoted to adopt these concepts to solve the problem of ranking Semantic Web data.

3.5.1. Query-independent Ranking

Studies such as HITS [Kle99] and PageRank [BP98] have shown that the notion of “authority” can help to distinguish Web pages. The ranking of Web results today are greatly influenced by the authority of pages. This metric is derived from the topology of the Web graph. That is, links between Web pages are identified and incorporated into the authority computation. Basically, pages that are well-connected get a high rank.

Semantic Search engines which rely on this notion of authority for ranking includes Swoogle [DPF+05], Sindice [ODC+08] and Falcons [CQ09]. Swoogle supports the search for ontology as well as classes and properties contained in them. It uses a PageRank-like algorithm for ranking these artifacts. Reconrank [HHD06] extends the concept of HITS to construct a global ranking of all Semantic Web data sources. That is, only the links between sources are considered but not the links at the level of data elements. A faster version of Reconrank has been implemented by Sindice. It requires less metadata to be collected and processed for each source. Along the same line, the authors of the work in [BM04] propose to use spreading activation for ranking answers obtained for structured queries.

Authority is certainly only one metric for estimating the relevance of answers. For NAGA [KSI+08], other factors such as the extraction confidence have been used. This is a measure that basically, stands for the
quality of the semantic data that has been extracted from textual data. It has been assumed that this measure of data quality has an impact on the relevance. Data with higher extraction confidence gets a higher rank.

Besides these approaches for ranking ontologies and answers, there are also studies devoted to the problem of ranking queries. This is important when the results returned by the systems are not directly answers but structured queries. Queries might be returned as possible interpretations of the keyword query the user has provided. Also, queries might be returned as possible expansions of the terms the user is current entering. Since queries in fact, can be conceived as sets of answers, the techniques for ranking answers can also be applied to deal with this problem. In [WZL+08, TWRC09], we have proposed to compute the importance of both nodes and edges in the semantic data graph. Instead of using PageRank, a measure based on the TFIDF concept commonly used in Information Retrieval has been applied. The intuition behind it is to consider those elements more important, which occur more frequently in the collection. The reason for not using PageRank or other authority-based concepts is because they become very expensive and unmanageable when dealing with authority not only at the level of sources but at the level of individual data elements. The number of the data elements is possibly very large, thus requiring complex computation. Further, since the edges between data elements have different semantics, the effectiveness of authority-based ranking heavily depends on the weights assigned to different edge types – a task that requires (upfront) manual effort that is not suitable for the large-scale Web setting.

3.5.2. Query-dependent Ranking

Query-dependent ranking is essential for Semantic Web Search engines like Sindice [ODC+08], Watson [SBG+06], Swoogle [DFJ+04] and Falcons [CQ09]. Essentially, these systems provide lookup functionalities based on an IR engine, such as Lucene. The IR engine is used to index ontologies, and the containing semantic data. Keywords submitted by the user are then matched against the indexed resources, where results are ranked according to the matching scores returned by the IR engine. In systems like Sindice, some additional ad-hoc rules are applied on top, e.g. “prefer data sources whose hostname corresponds to the resource’s hostname” [ODC+08].

For the problem of query ranking, the length of the computed queries
are used in [TCRS07] and the keyword matching score is employed in [ZWX+07].

For the multi-source scenario, we have implemented for the Hermes [TWH09] system a novel concept called EF/IDF (entity frequency/inverse document frequency) to combine the popularity with the distinctiveness of resources. Further, not only the data but also the rank of the sources containing them are taken into account. Data sources are ranked according to a query coverage measure. Intuitively speaking, sources are ranked high when they contain answers to a large part of the query.

Needless to say, not only query-dependent metrics but a combination of query-independent and query-dependent metrics are employed by state of the art Semantic Web Search systems to perform online computation of the query and result ranking.

3.6. Semantic Data Retrieval on the Web

Semantic Web Search engines discussed previously including Falcons [CQ09], Sindice [TDO07], Swoogle [DFJ+04] and Watson [SBG+06] focus on indexing and providing keyword-based lookup services. While a large number of data sources might be used, every result returned to the user is from one single source.

In [TWH09], we presented Hermes, the first solution towards Semantic Web Search that truely supports search in a multi-source scenario. Hermes is an infrastructure that enables integration and search over an open set of real life Web data sources. Instead of single-source results, answers returned to the user might combine data from several different sources. Users express their information needs using keywords, which are then translated by Hermes to federated conjunctive queries. These queries are evaluated against the underlying sources using independent “local query engines” and results are stitched together by the federated query engine. Fig. 3.4 depicts Hermes’s conceptual architecture and a query example. Components used offline are distinguished from components supporting online processes.

In Hermes, semantic data and sources are considered as data graphs. During offline graph data processing, different information are extracted from the data graphs and stored in specific data structures of the internal indexes. Firstly, the labels of data graph elements are extracted. A standard lexical analysis is performed on the labels, resulting in a set of terms. These terms are stored in the keyword index. If no schema information
is available, we apply summarization techniques to construct a schema graph for a given data graph. Schema graphs are stored in a *structure index*. For ranking support, scores are computed and associated with elements of the keyword and the structure indexes. Additionally, tools are employed to discover mappings both at the level of the data and at the level of the schema. The computed mappings are stored in a separate internal index called the *mapping index*. Together, the internal indexes are used to identify the semantic data elements matching a keyword, and to retrieve schema graphs and mappings.

The *online processing* of keyword queries over Web data sources can be decomposed into three main steps, namely keyword translation, federated query processing and local query processing. The input, the intermediate queries as well as the output for our example are shown in Fig. 3.4(a).

**Keyword translation** focuses on translating keywords to structured queries. This requires the steps of keyword mapping and top-\(k\) query graphs search. According to a query ranking scheme, the query graphs computed via this procedure are sorted and finally, presented to the user. The user then selects the ones that correspond to the intended information needs.

The selected queries might cover multiple data sources. During *federated query processing*, the query graph selected by the user is decomposed into parts that can be answered using one particular data source. These
parts are processed locally by independent RDF stores that are responsible for one specific set of sources. These stores evaluate the “subqueries” assigned to them and return back the intermediate results.

The indexing of data is performed using the IR-based technique already discussed in Section 3.4. The query translation employed by Hermes extends the approach we discuss in details in Chapter 5 to compute federated queries instead of single-source queries. Chapter 5 describes the rationales and the techniques needed for online query translation and also, provides detailed information on the offline steps of label extraction, analysis and scoring that are needed for that. For processing the subqueries locally, we equip our RDF stores with the structure-aware approach mentioned previously. This approach is elaborated in detail in Chapter 6. We will now continue with the aspects more specific to the problem of multi-source search, i.e, the ones of federated query processing and data integration.

### 3.6.1. Federated Query Processing

Work on querying Semantic Web data in a multi-source scenario has been published in [LWB08] and [QL08]. In [QL08], the authors proposed optimization techniques for ordering joins, which we also employ in the planning of federated query execution. The main difference between the approach we implemented for Hermes [TWH09] and this line of previous work is that we leverage pre-computed mappings to produce results that combine data from different sources. More specifically, Hermes makes use of data-level mappings to perform similarity join. Typically, the processing of similarity joins [KP07, SSS04] involves an expensive computation of similarities. Hermes simply retrieves the mappings from the index to perform standard join.

Basically, Hermes’s federated query engine firstly decomposes a query into parts such that each can be evaluated against one particular source. Then, these subqueries are routed to the RDF Stores responsible for the corresponding sources. For optimizing performance, a planner is employed to determine an appropriate order of query execution. Finally, the results obtained from the local query processors are combined to arrive at the final results – using the data-level mappings retrieved from the mapping index.
3.6.1.1. Query Graph Decomposition

In Hermes, a structured query is treated as a graph $g_q$ containing two types of edges: (1) intra-source edges connecting elements that ask for data of one source and (2) inter-source edges $e_i \in E^I_q$ connecting elements that ask for data from different sources. Based on this structure, query decomposition can be accomplished by simply omitting all $e_i$ from the query graph. The resulting query graph is a set of strongly connected components $g_{q_i}$ containing only intra-source edges. Each $g_{q_i}$ represents a partial query that can be evaluated against one single source $g_{D_i}$. Fig. 3.5 illustrates the decomposition of our example query into three parts: $q_1$ on Freebase, $q_2$ on DBLP, and $q_3$ on DBpedia.

3.6.1.2. Query Planning

Query planning concerns with the order of execution of the partial queries. For this task, an “abstract” query graph $g'_q$ is employed. Its vertices represent the partial queries and the inter-source edges $E^I_q$ constitute links between them. Given $g'_q$, query answering breaks down to two operations: (1) processing the vertices of $g'_q$ to obtain intermediate result sets (i.e., local query processing), and (2) combining the intermediate results along the inter-source edges. The optimal order of execution of these operations is estimated according to the optimization techniques proposed for RDF in [QL08]. In particular, statistics (e.g. about selectivity) are collected to arrive at estimates for (1) prioritizing query vertices that more likely lead to smaller intermediate result sets and (2) selecting a cost-efficient join implementation given two intermediate result sets. With respect to
Table 3.2. Intermediate result sets for $q_1$ and $q_2$.

<table>
<thead>
<tr>
<th>q1 on Freebase</th>
<th>m5</th>
<th>q2 on DBLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>z</td>
<td>y</td>
</tr>
<tr>
<td>Stanford</td>
<td>uni1</td>
<td>per1</td>
</tr>
<tr>
<td>Stanford</td>
<td>uni2</td>
<td>per8</td>
</tr>
</tbody>
</table>

Table 3.3. Intermediate result sets for $q_2$ and $q_3$.

<table>
<thead>
<tr>
<th>q2 on DBLP</th>
<th>m6</th>
<th>q3 on DBpedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>y'</td>
<td>x</td>
<td>dbp:Person</td>
</tr>
<tr>
<td>per2</td>
<td>pub1</td>
<td>per3</td>
</tr>
<tr>
<td>per9</td>
<td>pub2</td>
<td>per7</td>
</tr>
</tbody>
</table>

the example illustrated in Fig. 3.5, vertices of $g_q'$ are simply $q_1$, $q_2$ and $q_3$.

3.6.1.3. Query Result Combination

The results obtained from the local query engines are combined to arrive at the final answer for the federated query. Each result set for a partial query graph $g_{q_i}$ can be seen as a relation $R_{q_i}$, where a column $r_i$ in $R_{q_i}$ captures bindings for a particular vertex of $g_{q_i}$. Table 3.3 shows three relations obtained for our example queries, i.e., $R_{q_1}$, $R_{q_2}$ and $R_{q_3}$ for $q_1$, $q_2$ and $q_3$, respectively. The relations $R_{q_i}$ are joined along the inter-source edges, i.e., $R_{q_i} \bowtie_{e_i} R_{q_j}$, where $e_i \in E^q_I$ connects $r_i$ (denoting a column in $R_{q_i}$) with $r_j$ (denoting a column in $R_{q_j}$). Two types of joins are distinguished in this context: (1) one is for processing query edges representing mappings between classes (2) and the other is for processing edges representing relation or attribute mappings. **Processing of Class Mappings** If $e_i(r_i, r_j)$ is a class mapping, i.e., $r_i$ and $r_j$ correspond to classes, a similarity join needs to be performed on the entities of $r_i$ and $r_j$. In order to perform this join more efficiently, entity mappings are pre-computed such that given a class mapping, a two columns “mapping relation” $R_m$ is retrieved from the mapping index. Such a relation contains pairs of entities that have been identified to match based on their similarity. Examples are shown in Table 3.3, i.e., $R_{m5}$ and $R_{m6}$ for the mappings between person $m5$ and $m6$. Using these results, the similarity join amounts to a two-ways join $R_{q_i} \bowtie_{e_i} R_{q_j} \bowtie_{e_j} R_{q_j}$ (we refer to as map join):

- the first relation is joined with the mapping relation (on the first column),
• and then, the resulting relation is joined with the second relation (on the second entity column).

With respect to our example, the operations $R_{q1} \bowtie_{r_y=r_{\text{PersonFB}}} R_{m5}$ $\bowtie_{r_{\text{PersonDBLP}}=r_y'} R_{q2} \bowtie_{r_y'=r_{\text{PersonDBLP}}} R_{m6} \bowtie_{r_{\text{PersonDBP}}=r_y''} R_{q3}$ have to be performed for the computation of the final results. Thus, the similarity join as discussed in literature [KP07, SSS04] is realized in our approach through two steps: (1) offline computation of mappings and (2) map join that exploits pre-computed mappings. This way, expensive online comparison of entities can be avoided.

**Processing of Relation and Attribute Mappings** The map join technique is also used for the processing of relation and attribute mappings, i.e., $e_i(r_i, r_j)$ connects two attribute or relation vertices $r_i$ and $r_j$. Technically, a relation mapping $e_i(r_i, r_j)$ can be treated as two class mappings $e_{i1}(\text{domain}(r_i), \text{domain}(r_j))$ and $e_{i2}(\text{range}(r_i), \text{range}(r_j))$ that express the correspondences between classes that are the domain of $r_i$ and $r_j$ and the range of $r_i$ and $r_j$, respectively. Correspondingly, the processing of a relation mapping breaks down to two map join operations,

$R_{q1} \bowtie_{e_{i1}(\text{domain}(r_i), \text{domain}(r_j))} R_{q4}$ and $R_{q1} \bowtie_{e_{i2}(\text{range}(r_i), \text{range}(r_j))} R_{q4}$.

The processing of attribute mappings is similar. However, only one map join operation is needed because an attribute mapping $e_i(r_i, r_j)$ expresses only the correspondence between the domain of $r_1$ and $r_2$.

Note that the processing of two intermediate result sets $R_i$ and $R_j$ results in all combinations of tuples in $R_i$ and $R_j$ that are similar on one entity. This captures the intuition that (complementary) information from two data sources should add up. With respect to our example, tuples are joined along entities of the type person. This results in a combination of different information about person, i.e., publication, employment and prizes.

3.6.2. Data Integration

Clearly, Hermes relies on mechanisms for computing the mappings, both at the data and at the schema level. One important concept implemented by Hermes is pay-as-you-go integration. This means that the system does not assume that all mappings are available beforehand. This assumption is not realistic in the Web scenario where data constantly evolves. Other systems that also rely on integration is Freebase\(^7\). However, unlike Hermes, Freebase copies data from other sources into a central system which

\(^7\text{http://www.freebase.com/}\)
then, manages the data as one single source. Freebase implements a centralized, albeit open and community-based Web database. Data from other sources are imported into Freebase in a controlled way. The problem of heterogeneity is alleviated by a centralized, manual integration and reconciliation effort (also called gardening). As opposed to that, Hermes is a truly multi-source search system in that it treats sources as separate resources, and support search and result integration over these different sources. Instead of manual integration, it relies on techniques for the automatic computation of mappings.

We will now provide a brief survey on concepts and techniques proposed for data integration. Data integration is concerned with the identification of equivalent classes (schema-level elements) and instances (data-level elements). Correspondingly, approaches can be classified into two categories, namely (1) schema-level and (2) data-level integration.

### 3.6.2.1. Schema-level Integration

Schema-level data integration has received much attention from the database community in the past four decades. Representative systems include COMA [DR02], Protoplasm [BMPQ04] and S-Match [GYS07]. They follow the common paradigm of exploiting syntactical and relational features (structural information) of schema elements. Recently, instead of using one single technique, good results have been reported by hybrid systems that use combiners and multiple matchers.

The integration problem has often been formulated as a machine learning problem. Studies which discuss the use of machine learning techniques for integration can be found in [DDH01] and [DMDH02]. In these two studies, integration is treated as a classification problem.

In the Semantic Web community, much efforts have been invested to adopt schema matching techniques from the database community to solve the problem of ontology matching. Systems that have been developed for ontology matching include DSSim [NVVM07], RiMOM [TLL+06], Prior+ [MP07] and Falcon-AO [HQC08].

### 3.6.2.2. Data-level Integration

Data-level integration is investigated by different communities. It is known under different concepts such as record linkage [NK62], deduplication [SB02], reference reconciliation [DHM05]. Popular tools produced by database researchers for solving this problem includes TAILOR [EEV02], BigMatch [Yan02], MOMA [TR07] and Swoosh [BGMM+09] have been
developed. A recent survey on this problem can be found in [EIV07]. The above mentioned entity resolution tools all follow the single-global-threshold paradigm. That is, results outputted by the system are pairs of elements which have matching values that are larger than a global threshold.

In contrast to them, Chaudhuri et al. proposed in [CGM05] two novel criteria that can capture the local structural properties of instances and enable more accurate characterization of duplicated records. The basic idea of this work is to detect equivalent instances according to relative local distances to neighboring elements, rather than absolute global distances.

In the Semantic Web community, data-level data integration has not attracted much attention until recently – as the amount of available data has increased. In [NUMR08], Nikolov et al. presented a correferencing approach, which has been implemented as part of the KnoFuss knowledge fusion architecture. This approach also features the reuse and combination of different methods. In [GdM09], Gracia et al. presented an approach to clustering senses on the Web of data. The approach is based on using a Semantic Web Search engine to group senses sharing certain keywords (i.e., to create synonym maps for senses), and to adjust so called integration level to speed up the clustering process within each group.

### 3.6.2.3. Efficiency

For the integration of data on the Web, efficiency and scalability are crucial aspects. For data integration and especially for data-level integration, a technique called “blocking” has been applied. The idea is to break down the data into smaller chunks called blocks. The goal is to reduce the number of candidates that have to be considered for mapping computation. Various blocking methods have been discussed in [EIV07]. Representative blocking techniques include sorted neighborhood [HS98], clustering [ME97], canopy [MNU00], and prefix filtering [CGK06].

Another strategy to speed up data integration is to improve the efficiency of similarity calculations. Studies in this category include the following ones: Cohen et al. [CKM00] use inverted indexes and an A* search algorithm to efficiently find the top-\(k\) most similar record pairs. Sarawagi and Kirpal [SK04] proposed the Probe-Cluster approach that enables faster calculation of a large number of similarity predicates using set joins.

Recently, Whang et al. proposed a framework for iterative blocking where the clustering result of one block is propagated to other blocks.
3.7. Conclusions

We have provided a state of the art survey of Semantic Web Search systems. We discussed the underlying techniques, from crawling the data to storing and indexing the data and to querying and ranking the data. Also, we have discussed search in a multi-source scenario. Semantic Web Search engines index large number of sources found on the Web. However, every result returned is from exactly one single source. By multi-source search, we refer to the kind of feature which allows search to be conducted over several sources such that a result might combine data from different sources. For this, we have presented our system called Hermes, which is the first of this kind that supports truly multi-source Semantic Web Search. Central to this kind of search is federated query processing and data integration. We have presented the procedure for federated query processing employed by Hermes, which heavily leverages precomputed mappings for the sake of efficiency. Approaches that help to automatically compute these mappings have been discussed.

Besides database techniques, IR technologies have been embraced for Semantic Web Search. The application of IR techniques is not exclusively focused on ranking – which is probably the most straightforward idea to deal with the Semantic Web Search scenario where results might vary in quality and especially, might be very large in size such that only the computation of the top ranked ones is affordable. Besides that, the inverted index commonly used for IR tasks have been adopted for supporting indexing and storing semantic data. We discussed the specific approach that adopt the inverted index to scale to a large amount of semantic data and to support richer queries that are needed for Semantic Web Search. We argued that for semantic data management, IR technologies constitute a viable alternative to conventional databases.

At the beginning, we have pointed out that the main challenges in Semantic Web Search involve data volume, heterogeneity and complexity. In terms of volume, it seems that IR techniques can help to scale to a large amount of data, as proven by current Web search engines. However, IR techniques are limited to the processing of simple bag-of-words models, i.e., semantic data and queries have to be treated as textual data. However, the value of semantic data comes naturally from the rich structure and semantics it entails. To exploit this, the vast body of database
research has also been taken into account and adopted for Semantic Web Search. Database techniques have been used to process and optimize complex queries that capture rich structure constraints. Also, they have been adopted and extended to deal with the multi-source nature of Semantic Web Search, i.e., to integrates data (data- and schema-level integration) and to combines results at the time of query processing (federated query processing).

To sum up, IR techniques scale to large volumes of data but can only be used for simple queries. Database techniques are valuable for dealing with complex information needs that involve rich and complex constraints and heterogeneous results. In our opinion, the future of Semantic Web Search lies in the combination of this two lines of work. For scaling to complex Semantic Web Search, database techniques shall be married with IR techniques to scale beyond the realm of Terrabytes, and to support computing ranked results that are relevant to the user.
Chapter 4

Supporting the Semantic Web Search Process

4.1. Introduction

So far, search has been discussed from a conventional viewpoint. This viewpoint has been captured by the general search model and the various extensions of this model that we presented in the previous chapter. This chapter introduces the readers to the second main part of the book. It elaborates on the concept of Process-oriented Semantic Web Search where search is considered from a process-oriented point of view. In particular, it presents the main concepts of SemSearchPro and briefly discusses how this approach implements the Process-oriented Semantic Web Search model presented in the previous chapter in Section 2.6.

SemSearchPro combines work targeting different aspects of Semantic Search. It is basically a compilation of individual pieces of work, extended with the missing bits to fill in the big picture of Process-oriented Semantic Web Search. It recognizes that for addressing complex needs, the entire search process associated with it has to be taken into account. An effective solution needs to go much beyond the matching of queries against resources.

SemSearchPro is a schema-agnostic approach to Semantic Search in the sense that it does not assume the existence of a schema. It makes use of a lightweight semantic model that can be automatically derived from the underlying data. This model is the central element of SemSearchPro. It is the basis for implementing, as well as the glue for integrating specific modules that are dedicated to the individual steps of the process. This
semantic model is exploited throughout the process, from query construction to query processing, to result presentation and to query refinement. In particular, it enables efficient translation of keywords to structured queries, thus allowing users to construct complex queries using simple list of keywords. It is used as a guide during query processing, to focus on relevant candidates, to prune query parts, and ultimately, to improve efficiency. Further, this model is the basis for designing accessible presentation elements, and for the automatic selection of appropriate ones, given the current results. Also for query refinement, it is used to derive facets, which act as a description of resources and queries.

With respect to the objectives and challenges related to data volume, heterogeneity and complexity discussed in the previous chapter, SemSearchPro makes the following contributions:

- It uses the IR-based indexes presented in Section 3.4 for storing and accessing large volumes of data.
- It employs semantic models as summaries of the underlying data. Operating against summaries instead of using the data can substantially improve the performance. More complex queries can be processed against a larger volume of data. This is true for both the tasks of translating keywords to structured queries as well as for processing the structured queries.
- It addresses complexity also from the perspective of the user. Specifying complex information needs is a difficult task. Likewise, the interpretation of complex results might require support. Considering the entire process helps to reduce complexity at different steps. In other words, SemSearchPro addresses not only the complexity of query processing but also the complexity facing the user during query construction and result inspection.
- It supports the retrieval and integration of heterogeneous data by means of the mechanisms for federated query processing and data integration presented in Section 3.6.

This chapter does not discuss the work on indexing and federated query processing. We refer the interested reader to the previous chapter for more details on these aspects of SemSearchPro. Instead, this chapter focuses on the use of semantics through the online search process, i.e., query construction, query processing and result presentation and refinement. The topics and overall structure of this chapter is illustrated in Fig. 4.1. The data and queries supported by SemSearchPro is discussed
in Section 4.2. Also, this section discusses the notion of semantics and the concrete semantic model employed by SemSearchPro. Section 4.3 elaborates on the different steps of the search process that are supported by this approach. Three systems targeting different real world scenarios are discussed in Section 4.4 to demonstrate how SemSearchPro can be applied and implemented in practice. The case study in Section 4.5 contains results related to the effectiveness, efficiency and usability of the SemSearchPro approach to Process-oriented Semantic Web Search. This chapter concludes in Section 4.6.

4.2. Data and Queries in SemSearchPro

We will now present the models underlying the SemSearchPro approach. We will firstly discuss the internal representation of the underlying resources. Then, we introduce the formal query model used by SemSearchPro to represent information needs. Finally, we elaborate on the lightweight semantic model employed by SemSearchPro.
4.2.1. System Resource Model

In SemSearchPro, resources are represented using a general graph-structured data model:

**Definition 8 (System Resource Model)** A system resource model is a graph \( R(V^R, L, E^R) \) (also called data graph) where

- \( V^R \) is a finite set of vertices. Thereby, \( V^R \) is conceived as the disjoint union \( V^E_R \cup V^V_R \cup V^C_S \) with \( E \)-vertices representing entities, \( V \)-vertices \( V^V_R \) stand for data values and \( C \)-vertices \( V^C_S \) stand for classes.
- \( L \) is a finite set of edge labels, subdivided by \( L = L_R \cup L_A \cup \{type\} \), where \( L_R \) are relation labels and \( L_A \) are attribute labels.
- \( E^R \) is a finite set of edges of the form \( e(v_1, v_2) \) with \( v_1, v_2 \in V^R \) and \( e \in L \). Moreover, the following types are distinguished:
  - \( e \in L_A \) (A-edge) if and only if \( v_1 \in V^E_R \) and \( v_2 \in V^V_R \),
  - \( e \in L_R \) (R-edge) if and only if \( v_1, v_2 \in V^E_R \),
  - and \( type \), a predefined edge label that denotes the class membership of an entity.

The presented graph-structured model is similar to that of RDF. The intuitive mapping from RDF to this is, resources correspond to entities and classes, properties to either relations or attributes and literals to data values. We have discussed different forms of semantic data, including ontologies, linked data and embedded semantic data. Because they are represented as RDF, semantic data of these types can be mapped to elements of the presented resource model.

Note that \( V^C_S \) stands for a set of entities. It is in fact an element of the semantic model (hence we using the superscript \( S \) instead of \( R \)), as discussed in the subsequent section. It is also included in the resource model to recognize the fact that often, RDF data found in the Web comes together with class information. Clearly, the resource model does not require the existence of class information such that the set of \( C \)-vertices might be empty.

A general graph-structured model like this one captures not only RDF but also web documents, XML as well as relational data. In particular, this model subsumes the tree-structured XML data model. Relational data can be represented as graph-structured data, e.g. by mapping relations to vertices and foreign keys to edges. Likewise, web documents can be represented as vertices and hyperlinks connecting them can be modeled as
edges. Note that in fact, resources denoted by $V^R_E$ might be any real world objects. Documents are not explicitly distinguished from other types of entities. Just like other entities, documents might have relations to other entities such as author, publisher and special attributes such as title and abstract. This is illustrated in the example where artic2 is also represented as a E-vertex. Thus, documents can be indexed and managed just like entities.

**Example 2** An example data graph describing several entities is illustrated in Fig. 4.2. In particular, it says that per2 with name John McCarthy works at Stanford University, is author of artic2 and won the Turing Award. In this example, class information is available such that per2 is known to be of the type Person, uni1 is of the type University etc.

### 4.2.2. System Query Model

Internally, the information need of users are represented as a particular type of conjunctive queries, defined as follows:

**Definition 9 (Conjunctive Query)** A conjunctive query $Q_S$ is an expression of the form $(x_1, \ldots, x_k) \exists x_{k+1}, \ldots, x_m. P_1 \land \ldots \land P_r$, where $x_1, \ldots, x_k$ are called distinguished variables, $x_{k+1}, \ldots, x_m$ are undistinguished variables and $P_1, \ldots, P_r$ are query atoms. These atoms are of the form $p(v_1, v_2)$, where $p$ is a predicate symbol, $v_1, v_2$ are variables or, otherwise, are called constants.
Since variables of a system query $Q_S$ can interact in an arbitrary way, $Q_S$ is graph structured: it represents a graph pattern $Q_S = (V_{\text{var}} \cup V_{\text{con}}, L^Q, E^Q)$ consisting of a set of triple patterns $l(v_1, v_2)$ where $v_1$ and $v_2$ might be variables or constants, i.e. $v_1, v_2 \in V_{\text{var}} \cup V_{\text{con}}$.

**Example 3** An example query is $(x, y, z, u). \text{prizes}(y, z) \land \text{label}(z, \text{Turing Award}) \land \text{author}(y, u) \land \text{type}(u, \text{Article}) \land \text{employment}(y, x) \land \text{name}(x, \text{Stanford University})$. This query is illustrated in Fig. 4.3. It asks for $y$ working at place $x$ called Stanford University, which author articles $u$ and have won a Turing Award. All variables are distinguished such that all bindings to $x$, $y$, $z$, $u$ will be returned as answers.

Note that this query model is a restricted type of general conjunctive queries, a form of first-order queries. As opposed to the general formalism introduced in Section 2.4, query atoms are atomic formulas that draw exclusively from the set of binary predicates. The general model is well-studied in the literature, as it is capable of expressing a large portion of relational queries (relational algebra). The vast majority of query languages used in practice fall into this fragment, including large parts of SQL. The specific model used here has practical relevance because it is computationally more tractable. Also, the correspondence to SPARQL basic graph patterns as discussed in Section 2.4 holds for this specific type of conjunctive queries. We will now discuss the three common information needs that can be represented with this model:

- **Entity Search** In the IR community, this is also commonly known as navigational search. It is typically used as an entry point to the system, which is an entity such as a product, a Web page or a document in the collection. The user already knows the existence of the entity and uses the search function as a shortcut to this. Since the result is more or less known, the information need is expressed...
rather precisely, often with terms particular to the entity. The part \((x).\exists x.\text{name}(x, \text{Stanford University})\) of the example query for instance, represents by itself a query referring specific to an entity that has \text{name Stanford University}.

- **Fact Search** This refers to situations where the user is interested in a certain fact, like a phone number of a friend or the current temperature in San Francisco. While entity search involves one or several entities as results (E-vertices), this kind of search produces facts in the form of specific attribute values (V-vertices). Also, it is different to entity search in that it is not a navigational search for a known item, but rather, the purpose is to find unknown information. Thus, it is also referred to as informational search.

- **Relation Queries** This is another type of information search, where the goal is to gather not only information about a specific entity, but to find out complex set of entities, and especially, how they are related. The query example discussed previously belongs to this type, asking for relations between \(x, y, z,\) and \(u\).

### 4.2.3. Semantic Model

We will now discuss the semantics of queries and resources. The proposed query model is a fragment of conjunctive queries, which in general belongs to the class of first order queries. Thus, the queries have standard first order semantics. Precisely, every query is interpreted as a first order formulae that is constructed from atomic formulae using conjunction and existential quantification.

The formal semantics of resources can be established by mapping elements of the resource model to first order logic. Since the semantic model we are interested in refers to the conceptual part, rather than the instances, we will now focus on the conceptualization of resources and discuss its formal first order semantics.

The conceptualization of resources is built upon the basic notion of classes, class attributes and class relations. Basically, a class denotes a group of instances that commonly exhibit the same types of relations and attributes. This conceptual knowledge about entity types can be explicitly defined in the following semantic model:

**Definition 10 (Semantic Model)** A semantic model is a graph \(S(V^S, L, E^S)\) where
Chapter 4. Supporting the Semantic Web Search Process

Figure 4.4. A semantic model.

- $V^S$ is a finite set of vertices. Here, $V^S$ is conceived as the disjoint union $V^S_C \cup V^S_D$ with $C$-vertices $V^S_C$ representing classes and $D$-vertices $V^S_D$ stand for data types.
- Just like in the resource model, $L$ is a finite set of edge labels, subdivided by $L = L_R \cup L_A$, where $L_R$ are relation labels and $L_A$ are attribute labels.
- $E^S$ is a finite set of edges of the form $e(v_1, v_2)$ with $v_1, v_2 \in V^S$ and $e \in L$, where
  - $e \in L_A$ (A-edge) if and only if $v_1 \in V^S_C$ and $v_2 \in V^S_D$,
  - $e \in L_R$ (R-edge) if and only if $v_1, v_2 \in V^S_C$

Example 4 The semantic model for the data graph of our example is illustrated in Fig. 4.4. It captures persons, universities, articles and relations between them. University, Person and Prize have name and label that are of the type Text.

The formal semantics of this model is established by an explicit mapping to first-order logic. So far, edges $e(v_1, v_2)$ are only syntactic entities. They are given meaning by declaring that $e(v_1, v_2)$ holds exactly when this, taken as first-order sentence, is evaluated as true. Thus, we give the proposed model a first-order semantic by interpreting edges as first-order sentences:

Definition 11 (Semantics) A model $S(V^S, L, E^S)$ is given first-order semantics though the mapping of edges $e(v_1, v_2) \in E^S$ to atomic first order formulae $p(t_1, t_2)$, where $t_1, t_2$ are terms and $p$ is a binary predicate symbol.

Note that with respect to the different types of semantic models discussed in Section 2.3.3 of Chapter 3, the one presented here corresponds to a lightweight ontology. Its degree of expressiveness is less than RDF(S). It has been chosen as the central model for Process-oriented Semantic Web Search in SemSearchPro for the following reasons:
• **Tractability** Compared to the more expressive languages such as many description logics fragments defined for OWL2 and full first order logic, the model presented here is rather “lightweight”. While it has been associated with a formal semantics, the kind of reasoning that can be performed on it is limited, i.e., there is no implicit knowledge that can be inferred. We will see later that thus, matching query to resources simply involves the use of explicitly asserted knowledge. While reasoning capabilities can help increasing the expressiveness of search, and thus, satisfying more complex information needs, it comes at the cost of higher computational complexity. Since the kind of large scale and domain independent type of search we are concerned with typically involves a large amount of resources, reasoning is expensive. For online tasks and especially for search, this is still a problem because timely response is critical for user acceptance.

• **Generality and Extensibility** It is a basic model that is sufficiently general to conceptualize different kinds of resources, including documents of different types (Web, XML) and real world entities. Besides acting as a conceptual model, the basic first order semantics defined for it is also compatible to linguistic models, i.e. words can be interpreted as terms and linguistic relations between them mapped to predicates. Further, since this model and the semantic web languages RDF(S) and OWL rest on the same foundation, i.e. the one of first order logic, it is straightforward to extend it with additional modeling constructs. For instance, there are applications where knowledge about class hierarchies is available and can be effectively exploited for Semantic Search. In this case, formal conceptualization of the RDF(S) vocabularies `subClassOf` and `subPropertyOf` might be added.

• **Manageability** Most importantly, we note that for search, it is a difficult task to obtain conceptual knowledge. For instance, there is a large amount of RDFa data available, which represents useful information about the documents it is associated with. However, this data does not come with a schema. Also, for a large number of ontologies currently indexed by current Semantic Web search engines, the associated schema is incomplete, has poor quality or even does not exist. Further, resources under consideration are heterogeneous and evolve dynamically over time. Thus, the manual maintenance of the conceptual knowledge poses another problem. Using a more
lightweight semantic model solves this problem. This is because there are techniques, for which such a semantic model can be automatically computed from the data.

4.3. Process-oriented Schema-agnostic Search With SemSearchPro

This section discusses how the semantic model introduced previously is used throughout the process. Firstly, we describe this process in Section 4.3.1. Then, we discuss how the semantic model can be derived from the data in Section 4.3.2. The use of this model for query construction, query processing, result presentation and query refinement is then discussed in the subsequent sections. Note that this chapter focuses on how SemSearchPro uses semantics throughout the search process. Technical details related to the steps of query construction and query processing can be found in Chapter 5 and Chapter 6.

4.3.1. The Search Process in SemSearchPro

In this section, we discuss the idea of Process-oriented Semantic Search behind our approach called SemSearchPro. This approach instantiates the Process-oriented Semantic Search model introduced previously. The high-level overview on the individual steps involved in the search process as well as main the models employed for this are illustrated in Fig. 4.5. Central to this approach is the graph-structured semantic model. This conceptualization of resources and information needs might be established by hand, i.e., developed by knowledge engineers and made available in the form of a schema. As an alternative (or in combination with this traditional knowledge engineering approach), the semantic model we focus on can also be automatically derived from the data. Computing the semantic model from data is important especially on the Web, where schema information does not always exist for the available data, is incomplete or generally speaking, is costly to develop and to maintain.

The construction of the semantic model is performed offline. We will now discuss the online search process and the specific steps supported by SemSearchPro.

1. **Keyword Query**: The process starts with the information needs of users being represented in terms of keywords. The use of keywords as the user query model is driven by its popularity and widespread adoption. Past experiences with Web search engines have shown
that the keyword search paradigm is a simple and intuitive paradigm for expressing information needs.

2. **Keyword Translation**: However, the query model used internally is more expressive than that. In order to facilitate the construction of expressive information needs, the system automatically translates the keyword query entered by the user to a list of candidate conjunctive queries.

3. **Query Visualization**: These queries, representing possible interpretations of the keywords, are presented using different visualization modules, i.e., the queries can be visualized as graphs, presented as NL questions or as facets. Basically, facets correspond to predicates in the queries or in other words, represent descriptive elements of the current result set.

4. **Graph Matching**: The system retrieves results by processing the intended query (i.e., the one chosen and refined by the user). This amounts to matching the query graph against the data graph.

5. **Resource Visualization**: Just like queries, results are presented using different presentation modules. We distinguish generic presentation modules based on using graphs, trees and tables from customized widgets that take the semantics of the underlying resources into account. For instance, there are widgets for descriptions of people, events etc.

6. **Query Refinement**: Based on descriptive facets, the user can expand or refine the initially computed system query and in this way,
manipulates the current result set as needed.

Clearly, these steps have not to be executed in strict sequential order. Query refinement might be performed directly after query visualization, if the user so desires.

Using this approach helps to unfold the power of the semantics – which might be given explicitly or automatically derived from information only implicitly captured by the data. In particular, users can fully exploit the semantics for addressing complex information needs, which include up to complex relation search based on matching graph patterns. To do that, users do not have to cope with the internal representation of the resources and queries but instead, interact with more intuitive interfaces for constructing and refining queries, and for analyzing the results. During this process, the underlying semantic model is used not only for matching but also for implementing the translation and the presentation framework.

We will now discuss the construction of the semantic model and its usage during the search process in more details.

4.3.2. Offline Pseudo-Schema Construction in SemSearchPro

Recall that the semantic model in SemSearchPro is basically a conceptualization of resources based on the notion of classes, where a class denotes a group of instances that commonly exhibit the same types of relations and attributes. Typically, knowledge engineers define such classes based on their knowledge about the domain. In fact, such a semantic model can be designed beforehand in the form of a data schema. Then, data is put into the system to populate the schema. While this traditional workflow is commonly used for applications that are based on well-defined domain specific data, it is no longer applicable to large scale Web applications that have to deal with dynamically evolving generic data. In such scenarios, a schema cannot be defined completely a priori but must also evolve with changes in usage requirements, and with changes in the underlying data.

Instead of defining the conceptualization of resources a priori, SemSearchPro computes it automatically from the data to support a more affordable approach to Semantic Search. Note that in the SemSearchPro’s resource model, relations and attributes of instances represent edges of a data graph. Based on this observation, two strategies are employed to derive a pseudo-schema from the data, serving as the SemSearchPro’s semantic model. One strategy is simply to identify distinct classes, relation
and attributes mentioned in the data, and to construct the schema based on these elements. These two tasks can be achieved in one single run through the data, by means of clustering. The second and more sophisticated strategy is based on bisimulation. Basically, by running a bisimulation on the data graph, entities sharing similar structures can be found. Every group of such entities represents a schema element. These elements are put together to construct the pseudo-schema, i.e., the semantic model of SemSearchPro. These two strategies in fact lead to two different semantic models, which have different characteristics. While the one generated using the simple strategy is used for query translation, the other computed using bisimulation fully preserves the structure of the original data graph, and is thus exploited in SemSearchPro for the structure matching task performed during query processing.

4.3.2.1. Pseudo-Schema Computation via Clustering

A semantic model as defined previously basically describe classes, their attributes and relations between them. Attribute and relation labels in the semantic model are the same as the labels used in the resource model. Further, entity vertices in the resource model are often associated with class information in the case of RDF. Thus, there is sufficient information in the resource model that can be used to derive a kind of semantic model that is referred to as the clustering-based semantic model:

**Definition 12 (Clustering-based Semantic Model)** A clustering-based semantic model of a resource model \( \mathcal{R}(V^R, L, E^R) \) is a semantic model \( \mathcal{S}(V^S, L_R, E^S) \) where every class vertex \( v^S_C \in V^S_C \) represents an aggregation of entity vertices \( v^R_E \in V^R \) having the type \( v^S_C \), i.e., \( [v^S_C] = \{ v^R_E \mid \text{type}(v^R_E, v^S_C) \in E^R \} \) and the predefined class vertex \( \text{Thing} \in V^S_C \) represents the aggregation of all the vertices in \( V^R \) with no given type, i.e., \( \text{[Thing]} = \{ v^R_E \mid \exists v^S_C \in V^S_C (\text{type}(v^R_E, v^S_C) \in E^R) \} \). Accordingly, we have \( e(v^S_{C_1}, v^S_{C_2}) \in E^S \) if and only if there is an edge \( e(v^R_{E_1}, v^R_{E_2}) \in E^R \) for some \( v^R_{E_1} \in [v^S_{C_1}] \) and \( v^R_{E_2} \in [v^S_{C_2}] \).

This is a special semantic model that contains only class vertices \( V^S_C \) connected by relation edges with labels \( L_R \). An example of this model and how it is used for query translation is presented in Section 4.3.3. There is a correspondence between this model and the resource model from which it is derived: for every path in the resource model that contains only relation edges, there is at least one corresponding path in the clustering-based semantic model. This is however not true for the other way around. That
is, for some paths in the semantic model, there might be no corresponding paths in the resource model.

This can be easily seen as follows: assume the existence of a function $f$ which maps every entity vertex in the resource model to a set of corresponding vertices in the semantic model, i.e., $f : V_R^E \mapsto 2^{V_S^E}$. This function is defined on the basis of the above aggregations, i.e., $f(v_R^E) := \{v_S^C | v_R^E \in [v_S^C] \}$ and thus maps an entity vertex to all the class vertices representing its possible types. Now, it is clear that given an edge $e(v_{E_1}^R, v_{E_2}^R) \in E_R^E$ in the resource model, there is at least one (but possibly more) “corresponding” edges $e(v_{C_1}^S, v_{C_2}^S) \in E_S^S$ in the semantic model for some $v_{C_1}^S \in f(v_{E_1}^R)$ and $v_{C_2}^S \in f(v_{E_2}^R)$. Naturally, this notion of a corresponding edge carries over to paths, i.e., for sequences of the form $e_1(v_{E_1}^R, v_{E_2}^R), ..., e_n(v_{E_n}^R, v_{E_{n+1}}^R)$ in the resource model, there are corresponding paths $e_1(v_{C_1}^S, v_{C_2}^S), ..., e_n(v_{C_n}^S, v_{C_{n+1}}^S)$ in the semantic model.

The computation of this model follows straightforwardly from its definition and is accomplished by a set of aggregation rules, which compute the equivalence classes $[v_S^C]$ of all nodes belonging to one class $v_S^C$ and project all edges of the resource model to corresponding edges of the semantic model:

1. Every entity vertex $v_R^E \in V_R^E$ is clustered to the class vertices $v_S^C$ if there is an edge $type(v_R^E, v_S^C)$, If there is no such $v_S^C$, $v_R^E$ is clustered to $Thing$. During this clustering process, $v_R^E$ is deleted from the resource graph, $v_S^C$ is created and added to the semantic model when not already exists, and $v_S^C$ inherits all the edges from $v_R^E$ except type.

2. An edge $e_i(v_{C_1}^S, v_{C_2}^S)$ is clustered to the edge $e_j(v_{C_1}^S, v_{C_2}^S)$ if $e_i = e_j$. As a result, $e_i$ is deleted from the semantic model.

4.3.2.2. Pseudo-Schema Computation via Bisimulation

We propose bisimulation as the second strategy for computing groups of instances that are similar w.r.t. to the types of edges they exhibit. The concept of bisimulation originates from the theoretical analysis of state-based dynamic systems. In particular, we consider two graph vertices $v_1, v_2$ as bisimilar (i.e., $v_1 \sim v_2$), if they cannot be distinguished by looking at their “neighborhood of edge labels”. In other words, two vertices are bisimilar when they share the same structure as captured by edges and their labels. Based on this notion of bisimilarity, classes of a semantic model can be computed by considering pairwise bisimilar elements. More specifically,
a bisimulation is applied on the data graph. As a result, nodes of the data graph are partitioned into classes, each containing a set of pairwise bisimilar elements. Edges are constructed to connect a pair of two classes for all relations that exist between elements contained by these classes. The classes derived from the data via this process are also called extensions. The resulting semantic model is defined as follows:

**Definition 13** Let $\mathcal{R}(V^R, L, E^R)$ be the resource model and $\sim$ a bisimulation representing an equivalence relation on $V^R$. Vertices of the associated bisimulation-based semantic model $\mathcal{S}(V^S_C, L^R, E^S)$ are exactly $\mathcal{R}$’s $\sim$-equivalence classes $V^S_C = \{ [v] \mid v \in V^R \}$, with $[v] = \{ w \in V^R \mid v \sim w \}$. Labels of $\mathcal{S}$ are exactly the labels of $\mathcal{R}$. An edge with a certain label $l$ is established between two equivalence classes $[v]$ and $[w]$ in $\mathcal{S}$ exactly if there are two vertices $v^* \in [v]$ and $w^* \in [w]$ such that there is an edge $l(v^*, w^*)$ in $\mathcal{R}$, i.e., $E^S := \{ l([v^*], [w^*]) \mid l(v^*, w^*) \in E^R \}$.

An example of this model and how it is used for query processing is presented in Section 4.3.3. Further technical details on this model, information on parameterizing the structural depth and labels in the computation of bisimilarity to reduce the model size as well as details on computing the bisimulation can be found in Section 6.3.4.

4.3.2.3. Comparison Between these Models and Related Work

Clustering operates only at the level of individual graph edges. That is, structures that connect two or more edges are not considered during the pseudo-schema construction. The resulting schema tells that there are certain classes of instances in the data, and there are some attribute and relation edges between them. However, it is not possible to make any predictions about the structure of the entities contained in the classes. In particular, the existence of a schema path of more than two edges does not guarantee that in the data, there are some corresponding entities that are related via the same path over more than two edges. In fact, because single edges with the same label are clustered, the fact that a class node has one particular type of edges does not guarantee that all entities belonging to this class also have this type of edges. For instance, the pseudo-schema might contain the edge $\text{prize}(\text{Person}, \text{Prize})$. At the level of the data, there might be entities of type $\text{Person}$ which do not have any prizes. Thus, the pseudo-schema computed via clustering is similar to the notion of schema in practice: it specifies the possible structures (attributes and relations) an entity might exhibit but does not make any guarantees.
In contrast, the semantic model computed via bisimulation preserves the structure in the original data. Entities in one extension cannot be distinguished by looking at their “neighborhood of edge-labels”. This means that given the existence of an extension $E$ and an edge with the label $l_i$ connected to this extension, it is possible to infer that every entity contained in $E$ must also have at least one edge labeled $l_i$. In the case of complete bisimilarity, this inference holds not only for single edges, but for neighboring paths of arbitrary lengths.

Clearly, the bismulation-based semantic model captures more structural information. As a result, it is also bigger and more expensive to compute than the clustering-based model. We will now discuss the use of these two models for query translation and query processing.

4.3.3. Query Construction in SemSearchPro

In SemSearchPro, users articulate their information needs using keyword queries. This is an intuitive and simple paradigm for interacting with the Web, because for searching, users do not need to know the formal query language, the underlying data or the schema. They can simply use their “own words” to express information needs of different types.

To this end, the specific problem to be solved is to translate the given user keyword query $Q_U = \{k_1, \ldots, k_i\}$ to a list of candidate interpretations in the form of conjunctive queries $Q_S = (x_1, \ldots, x_k)$. $\exists x_{k+1}, \ldots x_m. p_1(v_i, v_j) \land \ldots \land p_n(v_k, v_l)$. In terms of the Process-oriented Semantic Search model we presented, the functionality to support this shall be provided by the semantic translation framework. SemSearchPro implements this translation framework as follows: The translation basically involves three main tasks: namely (1) the construction of the search space called the query space, (2) top-k query graph exploration and (3) query graph ranking. This procedure is similar to the ones used for keyword search in databases. Typically, the search space employed for keyword search in databases is the data graph [HWYY07, KPC+05]. Keywords entered by the user are matched against elements of this search space. The search space is also used for the exploration of subgraphs, which connect the keyword matching elements. Such an exploration might be very expensive when the data graph is large. While these approaches compute the actual answers, SemSearchPro focuses on the translation and thus, is concerned with computing queries only. For this purpose, the semantic model is used as the space of pos-
sible interpretations of the keywords, i.e., the query space. Clearly, the semantic model is typically much smaller than the actual data graph.

4.3.3.1. Construction of the Query Search Space

In fact, the construction of the query space is performed in two separate phases, resulting in two main parts:

- **Keyword matching elements** In the first step during the online search process, elements which correspond to the keywords \( Q_U = K = \{k_1, \ldots, k_n\} \) entered by the user are computed. This is performed via \( f : K \mapsto N_K \), i.e., a matching function that maps keywords to sets of graph elements referred to as the *keyword matching elements* \( N_K \) (also called keyword elements), where \( N_K \subseteq V_R \cup E_R \). In other words, keywords are interpreted as constants represented by some data graph vertices \( V_R \) or as predicates drawn from the set of edges \( E_R \).

- **Semantic model** The goal of translation is not only finding keyword matching elements but moreover, to discover what they actually mean in combination. Technically speaking, the aim is to find out how constants and predicates found in the first step are connected. Since the underlying semantic model captures the different ways resources are related, it can be used to explore the different interpretations of the computed keyword elements. For the purpose of translation, it suffices to operate with the semantic model computed via clustering. Recall that the main difference to the bisimulation-based model is that it does tell all the possible structures that can be found in the data, but at the level of entities, it does actually not guarantee that some particular structures exist. In other words, given there exist some interpretations, then they can be found in this model. Vice versa, it is however not true: interpretations that can be computed using this model might not correspond to data elements. At the level
of data entities, there might be no structures that instantiate one particular interpretation, i.e., there are no results to the query captured by the interpretation. Thus, using this semantic model instead of the data graph (or a more fine-grained bisimulation-based semantic model) can be seen as a way to improve efficiency – at the expense that computed queries might produce empty results. SemSearchPro favors this tradeoff for two reasons: (1) at large scale and in particular on the Web, searching large data graphs poses major challenges in terms of efficiency and scalability. The clustering-based model is a more compact summary of the search space that can address these challenges. (2) Even queries with empty results might correspond to the intended user information need. That is, some interpretations might point at the gap between user information needs and available data, indicating “missing data”.

The semantic model and the query-specific matching elements are combined to construct the query space.

**Definition 14 (Query Space)** A query space $S_Q(S, N_K)$ comprises the keyword matching elements $N_K$ computed for a query $Q_U$, which when not already contained, are connected with elements of the semantic model $S(V^S, L, E^S)$.

**Example 5** Fig. 4.6 illustrates the semantic model computed from the data graph in Example 2 via clustering. Note that attribute edges are omitted because for translation, only relations connecting C-vertices have to be considered. This model is extended to obtain the query space shown in Fig. 4.7. The elements denoting keywords in the query $Q_U = “Article Stanford Turing Award”$ have to be taken into account. Keyword elements obtained through matching the user keywords against labels are Article, Stanford University and Turing Award. Article is already part of the semantic model. Stanford University and Turing Award have to be added to obtain the complete query space for $Q_U$.

### 4.3.3.2. Exploration and Ranking

Given the query space, query interpretation amounts to searching for the minimal query graphs. Such a graph is a subgraph of the query space which contains at least one representative keyword matching element for every query keyword and further, these elements must be connected.
Chapter 4. Supporting the Semantic Web Search Process

SemSearchPro employs a top-$k$ graph exploration procedure to find minimal query graphs. It starts from the keyword elements $N_K$ and iteratively explores the query space $S_Q$ for all distinct paths beginning from these elements. During this procedure, the path with the highest score so far is selected for further exploration. For scoring paths SemSearchPro incorporates (1) the popularity of graph elements (obtained via an adapted version of TF-IDF called $EF-IDF$), (2) the matching score $S_m$ of keyword elements (obtained via the imprecise matching of keywords against graph element labels) and (3) the length of the path. At some point, an element might be discovered to be a connecting element, i.e., there is a path from that element to at least one keyword element, for every keyword in $K$. The paths between the keyword elements and the connecting element are merged to form a query graph. The graphs explored this way are added to the candidate list. The process continues until the upper bound score for the query graphs yet to be explored is lower than the score of the $k$-ranked query graph in the candidate list, i.e., no candidates can have a better score than the $k$-ranked result.

**Example 6** Fig. 4.8 shows the query space where elements are associated with some scores. For the interest of space, labels of “non-keyword elements” are omitted (refer to Fig. 4.7 for complete labels). Based on the element scores, the path score is updated at every step. For instance, the score of the path from Stanford University to the vertex with $EF-IDF = 0.012$ is the aggregation $(0.8 \times 0.12) + 0.027 + 0.012$. These path scores are then used to prioritize the “direction” of the exploration. The exploration starts from the keyword elements Stanford University, Article and Turing Award, as shown in Fig. 4.8. The three different paths starting from these elements that have been iteratively explored during the top-$k$ procedure are depicted using different line styles. For the first time, these three paths meet at the vertex with $EF-IDF = 0.012$, i.e., this vertex is a connecting element. These paths are merged to form the query graph of our example (as shown in Fig. 4.3).
4.3.4. Query Processing in SemSearchPro

Queries supported by SemSearchPro are conjunctive queries which basically, correspond to the notion of basic graph pattern in SPARQL. That is, a system query $Q$ is graph structured, representing a graph pattern. Essentially, a solution to $Q$ on a data graph $R$ is a mapping $\mu$ from query elements to data graph elements.

Thus, in SemSearchPro, addressing information needs of the user amounts to the problem of graph pattern matching. Clearly, $\mu$ is a certain type of homomorphism (i.e., a structure preserving mapping\(^1\)) from the query graph to the data graph. This perspective of considering answers to a query as homomorphisms will be used in the following. SemSearchPro employs the bisimulation-based semantic model to improve the efficiency of the matching task.

Recall that this semantic model is actually derived from structural patterns found in the data. It contains the different structures exhibited by data graph elements. In other words, it preserves the structure of the data graph. Thus, it can be seen as a compact representation of candidate answer graphs, where instead of single data elements, vertices represent set of elements called extensions. This is illustrated using the following example.

Example 7 Fig. 4.9 shows an extended example for the data graph. By means of bisimulation, the semantic model shown in Fig. 4.10 can be computed from it. Note that data graph elements that share the same structure are grouped and represented as extensions in the semantic model. For

\(^1\)As usual, a homomorphism from a graph $Q = (V^Q, L^Q, E^Q)$ to a graph $R = (V^R, L, E^R)$ is a mapping $h : Q \rightarrow R$ such that for every $Q$-edge $l(v_1, v_2) \in E^Q$ we have an according $R$-edge: $l(h(v_1), h(v_2)) \in E^R$. 

Figure 4.8. Three paths through the query space and their scores.
instance the extension $E_4$ comprises $uni_1$ and $uni_2$, which as illustrated in the data graph in Fig. 4.9, are similar w.r.t. to all the incoming and outgoing edges. Clearly, all distinct paths in the data graph in Fig. 4.9 are captured by this semantic model.

Note that this example of a semantic model is conceptually similar to the ones shown in Fig. 4.4 and Fig. 4.6. Extensions correspond to classes, which are connected by relations.

![Figure 4.9. An extended example of the resource model.](image)

Similar to the concept of query space, which is used for query translation, SemSearchPro makes use of the underlying semantic model for the purpose of matching. In this regard, the semantic model is also referred to as the “answer space”. However, unlike the clustering-based semantic model or the standard notion of data schema, the bisimulation-based model used here exhibits certain structural properties that can be exploited. Whereas edges between two classes in a schema do not guarantee that individuals belonging to these classes are actually related, this guarantee is provided by the bisimulation-based model. Given two entity vertices $v_i$ and $v_j$ that are contained in the extensions $E_i$ and $E_j$, and there is an edge with the label $l$ connecting $E_i$ with $E_j$, then it must hold that $v_i$ and $v_j$ is also connected via at least one $l$-labeled edge. As a result of complete bisimulation, this structural guarantee applies not only to single edges but structures containing paths of arbitrary lengths.

With respect to graph pattern matching, this means that whenever there is a match of a query graph on a data graph, the query also matches on the answer space. Moreover, extensions represented by vertices of the answer space matches will contain the data graph matches, i.e., the answers to the query. Thus, instead of operating directly on the data graph,
Chapter 4. Supporting the Semantic Web Search Process

Figure 4.10. The semantic model as an answer space for the extended resource model shown in Fig. 4.9.

SemSearchPro firstly searches the semantic model:

- In the first step, the query is matched against the answer space, resulting in a set of answer space matches. They contain data elements that satisfy the structure constraints captured by the query.
- In the second step, SemSearchPro verifies that these data elements not only match the structure but also the elements and content constraint in the query, i.e., constants and distinguished variables. For this, data elements contained in the answer space matches are retrieved, and joined along the query edges.

Example 8 Fig. 4.11 depicts a query, which asks for authors $y$ working at Stanford University that have won a Turing Award. Further, $y$ should supervise some $u$ that is author of some $v$. The matching of the query graph in Fig. 4.11 on the answer space in Fig. 4.11 results in one single match $h = \{x \mapsto E1, y \mapsto E4, z \mapsto E7, u \mapsto E3, v \mapsto E5, Stanford University \mapsto E6, Turing Award \mapsto E8\}$. Through this matching, we know that elements in $E4$ work at some places $x$, have won some prizes $z$ and supervise $u$. Further, we also know that $u$ is author of some $v$. Next, we check whether elements in $E4$ match the elements mentioned in the query, i.e., they really work at Stanford University, and have won a Turing Award. For this, we retrieve data contained in the extensions $E6, E1, E4, E7$ and $E8$ and join them along the edges employment($y, x$), name($x, Stanford University$), prize($y, z$), label($z, Turing Award$).

Note that using the answer space, data is retrieved and joined only for the parts of the query, which contain constants and distinguished variables. The remaining parts represent structure constraints only such that further validation at the data level is no longer required. It is not necessary
Figure 4.11. An extended example of the query graph.

because due to step one, answer space matches of the query are already known to satisfy the structure of the entire query. This is illustrated using the following example.

**Example 9** Continuing with our previous example, we can see that there are two parts that contain no distinguished variables and constants, i.e., the paths \( \text{employment}(x, u) \) \( \text{author}(v, u) \) and \( \text{supervises}(y, u) \). These parts can be pruned away after step one as all data elements contained in answer space matches are already known to satisfy these structure constraints, i.e., elements in \( E_4 \) are already known to supervise some \( u \) that are authors of \( v \), and elements in \( E_1 \) are known to employ some \( u \) that are authors of \( v \), respectively.

### 4.3.5. Result Presentation in SemSearchPro

The formal models of resources and queries are sufficiently expressive to address different kinds of information needs. The goal of our approach is to enable users harnessing this expressiveness, without having to deal directly with the underlying formal models – at anytime during the search process. So far, we have shown that the underlying semantic model can be leveraged for supporting a more lightweight query interface based on keywords, and for a more efficient matching of queries against resources. In this section, we will show that also, semantics can be exploited for enabling users to interact with more intuitive presentations of queries and results.

#### 4.3.5.1. Result Presentation

In particular, the semantics of resources are used to map the internal representation of resources to presentation modules of different kinds. Recall
that results to a query are of the types (1) facts, (2) entities and (3) tuples of entities (i.e., entities, their attributes and relations between them). These types of results are presented using generic and entity and fact specific presentation libraries. The mapping of the internal representation of results $\mathcal{R}_S$ to presentation libraries $\mathcal{R}_P$ are as follows:

- **Generic Presentation Interfaces** For generic results of types facts, entities and entity tuples, we use the generic interfaces fact box, list and table. In particular, factual results containing several data values ($V_{VR}^R$) are mapped to a fact box. Results containing several entities ($E_{VE}^R$) are mapped to a list where every entity description is rendered using a separate row in the list. A table is used to present entity tuples where entities’ relations ($V_{VE}^R$) and attribute values ($V_{VR}^R$) and are mapped to table cells, and attribute and relation names ($L_A$ and $L_R$) are mapped to column labels. Note that unlike tables in databases, the ones used here might contain information about entities of several classes, and different relations between them. For instance, one table might contain people, places, events, some of their attributes, and relations between them. Therefore, it might be unclear to which entities some other entities or attribute values refer to. In addition to the column labels, one another layer is thus used to depict the connections between them, i.e., the connections between the entities and data values contained in the columns.

- **Fact-Specific Presentation Interfaces** for single facts of certain types, there exist specific presentation modules. For instance, facts of the types location and time are mapped to visualization modules specifically designed for rendering locations (on a map) and times (on a timeline). Presentation modules are not limited to the purpose of presentation but might contain elements supporting further interactions. For instance, facts of the type telephone number are mapped to widgets that display the number and also, support actions such as “store number” and “call” (e.g. using Skype).

- **Entity-Specific Presentation Interfaces** for single entities of certain types, i.e. $E$-vertices of some classes $V_{VC}^S$, there exists also specific presentation modules. Note that for entities, the units of presentation comprise not only of the entity identifier but its entire description, i.e., a set of relations and attributes value pairs ($L_R-V_{VE}^R$ and $L_A-V_{VR}^R$). Accordingly, entity presentation modules are also composites, which might be constructed using fact-specific presen-
tation modules. There are specific modules for rendering entities of
types people, place, event, publication and projects. For instance,
attributes of persons are presented using fact-specific presentation
modules as obtained via the mapping of facts, e.g. location, tele-
phone number etc.

4.3.5.2. Query Presentation

In the traditional search process, users typically issue a query, obtain re-
sults, and if these do not satisfy the information need, start over with is-
suing a new query. The approach we propose emphasizes the steps of
supporting users during query construction and iterative query refinement.
For these tasks, users make use of the presentation query model QP.

Intuitively speaking, queries are abstract descriptions of results, i.e.,
every query describes a set of results using the descriptive elements at-
tribute and relation predicates and might also, contain concrete values in
the form of constants. Since the underlying semantics is the same, the
different query types, i.e., fact, entity and relation queries, can also be
mapped to the kind of presentation modules used for result presentation.
For instance, relation queries are illustrated using tables, where variables
and constants are mapped to column labels, and predicates are used to
denote the connections between them (using the additional layer as dis-
cussed before). Further, the following generic mapping of entities to pre-
sentation modules are employed:

- **Graph-based Presentation** Recall that since variables might inter-
act arbitrarily, conjunctive queries form a graph. For a more visual
presentation, the query is mapped to a graph-based visualization. In
particular, query atoms are mapped to edges, the constituent parts,
i.e., variables and constants, are mapped to graph vertices and pred-
icate names are used as edge labels. Further, distinguished query
variables are highlighted visually.

- **NL-based Presentation** In a straightforward manner, queries are
also mapped to constructs of a natural language. In particular, query
atoms are firstly grouped by the first term, i.e., to obtain groups
of atoms that “describe” the same variable. Then, the set of dis-
tinguished variables x1, . . . , xn is mapped to construct of the form
“Search for x1, . . . , xn, where” and relation query atoms pr(vi, vj)
are mapped to “vi is related with vj via pr”, and the remaining at-
tribute query atoms pa(vi, vk) are mapped to “vi pa is vk”.
• **Facet-based Presentation** Facets can be seen as description elements of the current result set. The result set comprises substitutions of distinguished variables found during the matching. A facet is either a single predicate \( p_x \) or a predicate-constant \((p-c)_x\) pair which refer to a set of results that are bindings to the variable \( x \). Since the queries under investigation might describe results as complex sets of entities, i.e., substitutions of several distinguished variables, they might be mapped to sets of facets. Every such set represent one description that is associated with a particular set of entities (a particular variable, respectively). More precisely, atoms are grouped by the first term to obtain sets of description, one for every distinct distinguished variable that appears as the first term of a query atom. Every atom \( p(var_d,v_j) \) is then mapped to a facet \( p_{var_d} \) if \( v_j \) is a variable, otherwise \( v_j \) is a constant and thus, it is mapped to \((p-v_j)_{var_d}\).

4.3.6. Query Refinement in SemSearchPro

Refinements to the query may be needed for several reasons. The computed interpretations may not exactly match the information need. Also, the user may start out with a vague information need, not knowing exactly what he is searching for. For these cases, a presentation query model which intuitively reflects the semantics of the result set, facilitates the modification of results and the comprehension of the resulting changes.

In particular, a facet-based presentation model provides the means for the user to narrow down or expand the resources of interest according to their information need in an iterative way. In particular, the user can add, remove or edit the facets (i.e. change the predicate of the constant). These operations are transparently converted to changes on the underlying query. The query refined this way is immediately evaluated, and new results are presented without the user having to explicitly issue a new query.


Fig. 4.12 shows the interfaces that can be used for the main steps of the search process, illustrating how it is supported by systems implementing SemSearchPro. In particular, it shows the presentations of queries and results, illustrating the concepts discussed in Section 4.3.5. The keyword interface shown in Fig. 4.12(1) is similar to those of Web search engines.
Queries computed from the user’s keywords are presented using graph- or table-based visualization, as shown in Fig. 4.12(2,3). Query refinement is achieved through a faceted search interface, which is another form of query presentation. Fig. 4.12(4) shows a facet menu that allows relations to be added to or removed from the query. Fig. 4.12(5) shows a more advanced interface for faceted search which enables filtering of the result set by either choosing hierarchically ordered facets or by entering values directly in the text box. Presentations specific for entities show the relations and attributes of an entity as a graph, structured data combined with the textual description of the entity and even live data such as twitter messages about the entity, see Fig. 4.12(6,8,9). Maps and diagrams, as shown in Fig. 4.12(7,10), are examples of fact-specific presentations that display facts of specific types.

We will now discuss the systems behind this: (1) AskTheWiki, (2) Hermes and (3) the Information Workbench. These systems serve as demonstrators for process-oriented and schema-agnostic Semantic Search. They instantiate the Process-oriented Semantic Search model presented in Section 2.5 of Chapter 3 and implement the SemSearchPro approach presented in the previous section. All systems support the entire search process, but focus on different aspects. We refer the interested reader to the more complete descriptions of these systems available in the respective conference papers and technical reports [HHMT09, WPX+09, HEG+09].

The main goal of AskTheWiki\textsuperscript{2} [HHMT09] is to study search in the context of a specific information portal. The search system is developed on top of a system called Semantic MediaWiki (SMW) [KVV\textsuperscript{+}07]. It operates on the semantic data provided by this system in the form of RDF. This RDF data is transformed to the graph-structured resource model of SemSearchPro.

In SMW, users have two mechanisms for searching. They can type keywords, upon which a standard IR engine is used to retrieve results containing these keywords. For the technical users, SMW also supports structured queries, which have to be specified in the SMW-specific syntax. The main focus of AskTheWiki is to facilitate the task of query construction in SMW by means of query translation. Besides this, AskTheWiki implements the query presentation and refinement concepts of SemSearchPro to support the users throughout the search process.

4.4.2. Open Environment: Multi-Source Search on Linked Open Data

The Hermes system [WPX\textsuperscript{+}09] targets search in a multi-source scenario. It has been built to cope with the various datasets available for the Billion Triple Challenge\textsuperscript{3}. Most of this data is from the set publicly available sources published as linked data. About 1 billion triples have been chosen, compiled and made available as the Billion Triple Challenge (BTC) dataset\textsuperscript{4}.

Hermes operates on nearly 1 billion triples of the BTC dataset. It leaves out only some data sources that are small and contain only few links to other sources. Hermes supports querying and combining results from different sources in the BTC dataset using keywords. User keywords are translated to federated structured queries. Also, subsequent exploration and refinement is possible using faceted search. Especially for this setting, the use of a more compact semantic model has proved essential for efficient query translation and query processing. Besides the SemSearchPro’s online search concepts, specialized techniques and indexes for computing and managing mappings have been implemented for this system.

\textsuperscript{2}http://www.aifb.kit.edu/web/Spezial:ATWSpecialSearch Nov 25 2009
\textsuperscript{3}http://challenge.semanticweb.org/
\textsuperscript{4}http://challenge.semanticweb.org/
These mappings established links between data sources, which are used for federated query processing (the schema level mappings) as well as for the combination of results that come from different sources (the data level mappings).

4.4.3. Beyond Search: Interacting with Linked Open Data

The Information Workbench\textsuperscript{5}\cite{HEG09} targets not only search, but the entire process of interacting with the Web of Data. Users can import data, then search, explore, analyze as well as refine it, and finally, publish it back to the Web. Besides the query translation and processing functionalities that are similar to the ones also available in AskTheWiki and Hermes, this system features a concept called “Living UI”. Basically, it is an implementation of the SemSearchPro result presentation concept. It offers an adaptive user interface, where results are presented using type-specific widgets.

4.5. Process-oriented Schema-agnostic Search User Study

The work presented here brings together pieces of work targeting different aspects of the Semantic Web Search process. The evaluation results for query translation and matching are discussed in detail in Chapter 5 and Chapter 6. While these results concern the use of the semantic model for the individual steps, we have also conducted an experiment considering the Semantic Web Search process as a whole. The goal for this is to assess the applicability of process-oriented schema-agnostic search as implemented by SemSearchPro in a real-life scenario. We performed a user study to evaluate the system primarily in terms of effectiveness and efficiency, and also derived some initial results concerning user satisfaction and usability. This section discusses the results of this search process evaluation.

4.5.1. Evaluation Setting

Since SemSearchPro takes the entire process into account, the standard IR evaluation based on the technical measures of precision and recall does not apply. This paradigm is too limited to assess the overall effectiveness.

\textsuperscript{5}http://iwb.fluidops.com Feb 2 2010
and efficiency of the overall search process. Thus, we follow the task-based evaluation design that has gained acceptance in the IR community for evaluating interactive systems [ER07].

In particular, we chose a task-based user evaluation where each task represents an information need that typically occurs when using the portal. After the task evaluation, we asked the participants to answer a multiple choice questionnaire about their experience. The questions concerned their technical background and their experience and satisfaction with certain aspects of the supported search process.

Participants of the user study were 14 volunteers from four different organizations active in the Semantic Web community. The users know the semanticweb.org platform. While they partially know the kind of data that can be found there, they do not know the schema (as it is not explicitly available). All participants are familiar with keyword search. Some users (25 percent) do not know (how to use) SPARQL, the system query language $Q$ used in our implementation of process-based search.

Each participant was given five tasks and had up to three minutes to solve each task. A task could be skipped, if participants felt that they could not solve the task. The participants received limited information about the search interface upfront, namely that they do not need to know the schema and a formal query language, that the search process consists of three steps, that the system will not return a list of documents like common search engines but interpretations of their keywords, and that they have to choose an interpretation, and that the results can be modified in the third step. All actions taken by the participants and the system responses were logged. In particular, we logged the users’ steps, the keyword inputs as well as the system responses. We measured how often users could solve the task and how much time it took them. The evaluation was performed based on both the analysis of the log files as well as the questionnaire.

Additional information such as the questionnaire and the handouts provided to participants can be found at [HHMT08].

4.5.2. Tasks

Designing the tasks is crucial for the success of an evaluation. The tasks were constructed to assess the quality of the individual steps and more importantly the search process as a whole. They cover different levels of difficulty. In particular, we created tasks corresponding to queries that fall into the search categories introduced in Section 4.2, i.e., entity queries, fact queries and relation queries.
We created two task sets as shown in Table 4.1 and Table 4.2. Both sets have the same structure and cover the same query types.

<table>
<thead>
<tr>
<th>Task No</th>
<th>Task Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d</td>
<td>Find the wiki page of AIFB.</td>
<td>Entity query</td>
</tr>
<tr>
<td>2d</td>
<td>When is the paper deadline for the ASWC2008?</td>
<td>Fact query</td>
</tr>
<tr>
<td>3d</td>
<td>What is the email of Holger Lewen?</td>
<td>Fact query</td>
</tr>
<tr>
<td>4d</td>
<td>Find exporters with GPL license and their homepages.</td>
<td>Relation query</td>
</tr>
<tr>
<td>5d</td>
<td>Find the capitals of countries in Europe and the population of these cities.</td>
<td>Relation query</td>
</tr>
</tbody>
</table>

Table 4.1. Task set 1 for semanticweb.org.

<table>
<thead>
<tr>
<th>Task No</th>
<th>Task Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e</td>
<td>Find the wiki page of Stanford University.</td>
<td>Entity query</td>
</tr>
<tr>
<td>2e</td>
<td>What is the homepage of the ISWC2008 conference?</td>
<td>Fact query</td>
</tr>
<tr>
<td>3e</td>
<td>What is the email of Thanh Tran?</td>
<td>Fact query</td>
</tr>
<tr>
<td>4e</td>
<td>Find reasoners with GPL license and their homepages.</td>
<td>Relation query</td>
</tr>
<tr>
<td>5e</td>
<td>Who was the local chair of the conferences located in Karlsruhe in 2008?</td>
<td>Relation query</td>
</tr>
</tbody>
</table>

Table 4.2. Task set 2 for semanticweb.org.

4.5.3. System and Data

The evaluation was conducted using the *AskTheWiki* system. It was performed within the community portal *semanticweb.org*, a wiki-based platform serving the Semantic Web community. The wiki contains information about the Semantic Web such as events, publications, tools, and people. This data comprises a total of 55,365 triples (as of Dec 4 2008).\(^6\) The data was created by the users of the wiki over the last three years. Since the nature of a wiki is to provide unconstrained user editing, the data does not follow a predefined vocabulary or strict schema. Rather, it evolves with changes in the data. For the evaluation, we have computed a semantic model from the data. It has a size of 7,101 triples.

---

\(^6\)The data is available at [http://semanticweb.org/RDF](http://semanticweb.org/RDF).
4.5.4. Effectiveness

To measure the overall effectiveness, we analyzed the ratio of tasks that have been successfully completed. Overall, 6 out of 14 participants were able to fulfill all five tasks, 12 of the 14 were able to fulfill 60% or more. The other two users quickly gave up after the first or second task stating that they found the system too complicated (see Fig. 4.15). For the simple tasks (entity queries), the success rate was 100%, the more complex tasks result in lower success rates: 79% for fact queries and 64% for relation queries. These results are illustrated in Fig. 4.13.

![Figure 4.13. Effectiveness of search by query type.](image)

Fig. 4.14 shows the success rate for each individual task. There is a notable difference between the success rates of task 4e and 4d, as well as between 5e and 5d. For tasks 4e the success rate is comparatively low, because four of the seven participants did not enter the keyword “homepage”. One reason for this difference might that participants misunderstood the term “homepage” in the query 4e, and thought that the links to wiki pages of the reasoners with GPL licenses are already the results. For 4d, only one participant did not use the keyword “homepage” and therefore most users found the actual results, namely the homepage of the exporters instead of the wiki page. Task 5e has a lower success rate than 5d because many participants used the keyword “2008” stated in the task description. This keyword is problematic because it matches many elements in the data, i.e. many attribute values contain “2008”. Due to this kind of ambiguity, the quality of the generated interpretations were not high, and therefore this task yielded a lower success rate.

In conclusion, the search process was effective in supporting the tasks, as most of them could have been completed by all users. The problematic
cases are when users enter too few keywords, resulting in incomplete interpretations or when keywords are too ambiguous, resulting in too many matches and interpretations respectively.

### 4.5.5. Efficiency

To assess the efficiency, we measured how many search iterations were performed by counting the number of keyword queries a user had to reissue per task. The results are shown in Fig. 4.16 grouped by the types of queries and for each task separately in Fig. 4.17. We see that on average the users needed to issue between 1.6 and 2 keyword queries to fulfill a task, depending on the query type. Expectedly, the value is larger for the more complex relation queries (2 keyword queries per task) than for the simple types of queries. As shown in Fig. 4.17, there is large difference between the number of queries issued for tasks 5d and 5e. This is directly correlated to the low success rate of task 5e, which we just discussed. The
participants tried harder to solve 5e and thus issued more queries, whereas task 5d was apparently easier to solve.

Beside the total number of process iterations per task, we also measured the total process cost in terms of the steps taken and the time consumption. Note that users do not have to work through the entire search process. At any step, users might decide to stop when the information need is found to be fulfilled. For example, when searching for an entity, the returned interpretations might already contain the entity, and thus the answer. The actual process steps taken by the users are shown for tasks of different types in Fig. 4.18. We see that only 21% of the 24 search processes executed for entity search tasks continued to the second step, whereas in 40% and 61% of the fact and relation search processes respectively, users had to chose an interpretation. Refining the search result was performed in 17%, 11%, and 20% of the search processes respectively, as shown in Fig. 4.18.

![Figure 4.16. Efficiency of search by query type.](image)

We measured the time spent for a task by summing up the duration of all steps performed by the user for each task. The total duration for a search is the time difference between the keyword query arrival at the server and the last action performed by the user. The time needed by the user to actually type the keywords is not included in the measurement.

For the entity search tasks, the users needed on average 9.3 seconds with a median of 0.5 seconds. The median is significantly lower than the average, since 8 of the 14 participants needed just one keyword query to complete the task and did not take any further steps, as shown in Fig. 4.18. The fact search tasks took on average 12.8 seconds with a median of 9.6 seconds. The participants spent on average 52 seconds with a median of 44.5 seconds on the relation search tasks.
Further, we recorded the time needed for two main parts of this process: the time up to choosing an interpretation and the time needed for every subsequent refinement of the result set. Part 1 took the participants 14.7 seconds (median). Between refinement operations they participants spent 14.0 seconds (median). Note that part 1 includes the time needed by the system to actually compute the interpretations. We already discussed the performance results that can be achieved for this kind of processing. It actually makes up only a small share, while the rest of the time is needed by the user to understand and to choose the intended interpretation. The second part contains the time the system needed for query processing but also here, most of the time is actually spent by the user. More than 90 percent of this time is needed to understand the results and to assess whether they fit the information need.

The overall results are encouraging, showing that even tasks which involve complex structured queries and results, require no more than 2 process iterations, and less than one minute in total. Also, the overall success rate is encouraging, considering that the participants have varying technical backgrounds, do not know the underlying data schema, and used the system, which was new to them, without detailed usage instructions.

4.5.6. Usefulness

We now discuss the usefulness of different aspects of the implemented search process. This discussion is based on the results of the questionnaire. The responses to these questions are shown in Fig. 4.19.

**Articulation of the information needs** The first question asked how difficult the users found it to express the information need in keywords. As
expected, the users found it rather easy to do so, as all of them are familiar with keyword-based search interfaces. The results of the questionnaire tell that all participants use Google and some have tried other search engines with more advanced features such as Powerset and Cuil.

Translation Overall, the users found the representation of interpretations easily comprehensible, and it was easy for them to choose the right interpretation. Two users had difficulties (see questions regarding step 2 in Fig. 4.19). One reason was that in some cases the interpretations were so similar that the users could not easily tell the differences.

Presentation & Refinement The majority of the users found the presentation of the results understandable. However, only seven users made use of faceted search to refine a query. As answers to the question about how useful the feature was to the users, most participants found it very useful or useful to modify the interpretations, whereas three participants stated that they did not know how to do it (see questions regarding step 3 in Fig. 4.19). This suggests that when users know how, faceted search is useful in this process. Unlike keyword search, faceted search is however not yet a commonly used paradigm. Effective use might require more detailed instructions, which were (deliberately) not given in our experiment. Interestingly, the use of the faceted search was particularly effective for the more complex tasks. On average, 29.6% of the successfully completed tasks involved refinements using faceted search. For the most complex
Figure 4.19. Results of the usability study.

tasks involving relation search, 38.9% of the successfully completed tasks involved the use of faceted search. We thus have reason to believe that the overall success rate would have been higher, if all users had known how to utilize faceted search.

4.6. Conclusions

We discussed a compilation of our work called SemSearchPro, which implements the Process-oriented Semantic Search model. Besides, SemSearchPro can also be considered as a schema-agnostic search approach because it does not require the existence of a data schema. This is possible by using a lightweight semantic model that can be automatically derived from the underlying data.

This semantic model has been formally defined. It has been discussed in terms of tractability, generality, extensibility and manageability. It has been chosen as the central model of SemSearchPro because it lends itself to tractable processing, can be extended to incorporate more complex reasoning, is sufficiently general to capture different kinds of data on the Web and Semantic Web, and does not require manual maintenance.
The use of this semantic model throughout the entire process has been discussed. It is used as a compact summary for more efficient query translation. Along the same line, it has been used to improve the performance of query processing. Also, it is used as the basis for enabling adaptive result presentation and iterative refinement.

Three systems implementing SemSearchPro were presented to demonstrate how the search process can be supported in real world scenarios. We evaluated different aspects of SemSearchPro. The results for the individual steps suggest that the use of semantics can lead to an improvement in performance, for both the query translation and query processing tasks. The task-based evaluation, which considers the process as a whole, suggests that SemSearchPro is efficient and effective. Most of the tasks were completed in a reasonable amount of time.
Chapter 5

Query Construction and Refinement

5.1. Introduction

This chapter deals with the first step of the Semantic Web Search process. Unlike standard Web search where information needs are expressed using a simple list of keywords, Semantic Web Search aims to address more complex needs which cannot be easily captured using simple keywords.

In fact, the main motivation for using semantics in Web search is to deliver more precise results to more complex needs. Instead of documents, results finally returned to the user might range from precise answers in the form of facts, to resources and their relations up to integrated units of content. Such integrated units of results might contain semantic data embedded in documents or vice versa, textual data embedded in Semantic Web resource descriptions. Compare to documents, these types of results capture more complex structures and semantics, and more precisely match the user needs. To harness these advanced capabilities of Semantic Web Search, users have to specify complex information needs. This might be a task too complex to be performed by the typical Web users.

This chapter discusses approaches for query construction and iterative query refinement, which aim to address this problem. The focus of this chapter lies in a special type of approach, namely the schema-agnostic query interfaces. As opposed to the other approaches, this search paradigm does not require the user to know formal query languages and in particular, it does not require knowledge of the schema and the underlying resources. This is essential in the Web setting, given the goal of Web search is to lower the barrier of entrance. Also non-technical people shall be able to explore new domain and to search for Web resources that are previously unknown.
In Fig. 5.1, we illustrate the main topics and the structure of this chapter. To introduce the readers to the concept of schema-agnostic search, Section 5.2 discusses the state of the art of schema-agnostic query interfaces. In particular, this section covers (1) standard IR-style keyword search (on the Web), which computes simple “direct matches”, (2) keyword search in databases, which compute complex results comprising direct matches as well as connections between them (also called result completion), and (3) keyword query translation, where the goal is to output structured queries that capture the intended meaning of the keyword query (also called query completion). Further, (4) faceted search is presented in this section. This paradigm has become the principal mechanism for the iterative refinement of queries. Section 5.3 is dedicated to the concepts and techniques used in SemSearchPro. It demonstrates in detail how semantics can be used to facilitate query construction and query refinement. Experimental results on the efficiency and effectiveness of the existing approaches and the results obtained for the SemSearchPro approach are discussed in Section 5.4. This chapter concludes in Section 5.5.
Chapter 5. Query Construction and Refinement

5.2. Schema-agnostic Query Construction and Refinement Approaches

The standard type of search supported by Web search engines is based on the keyword search paradigm. Using keywords, the user finds the resource of interest (informational search) or often, obtains some results as starting points (navigational search) first, from which further information is then discovered and retrieved via additional browsing. This paradigm has proven to be intuitive as well as effective in addressing simple information needs. More complex needs however require users to try out different keywords, and to browse and navigate along the complex Web space.

With the rise of the Semantic Web and the large and increasing amount of semantic data it contains, there are new opportunities for addressing more complex information needs. We illustrate this with the following scenario.

Example 10 Mary is a novice computer science student at KIT. She is eager to learn more about this vast research field and decided to find information about research work of researchers at AIFB.

With traditional Web search, Mary searches and browses to find researchers at AIFB first. Then, another round of search and browsing is needed to find information about their work. Browsing is performed purely based on traversal along hyperlinks and keyword-based navigational search. The Web space is complex, containing large amounts of hyperlinks and Web pages that are relevant to a navigational query. Solving this task is thus time consuming. Using the Web of data, this complex information need can be addressed using one single query. For this, Mary specifies the need using a structured query language and obtains right away the precise answers. As discussed before in Section 4.2.2 in Chapter 4, the conjunctive queries used in SemSearchPro represent one language that supports various types of information needs ranging from simple entity search to fact search to complex relation search that can be used for gathering complex information (i.e., n-ary tuple sets of results).

The scenario above is one example for relation search. It illustrates that complex information needs can be addressed more efficiently using Web data. However, it also makes clear that users face the burden of specifying complex needs. Constructing structured queries requires the user to know the language and more importantly, the schema as well as the data. In this case, Mary needs to know the schema elements researchWork,
researchers, worksAt, name and the data element AIFB to specify the structured query \( (y, z) \cdot \text{type}(y, \text{researcher}) \land \text{worksAt}(y, x) \land \text{name}(x, \text{AIFB}) \land \text{researchWork}(y, z) \). Especially in the context of querying Web data, this is too large a burden for the users. The data as well as the schema are evolving, making it difficult for users to keep track of the changes and to be always knowledgeable. Moreover, search on the Web often involves unknown resources. In this case, the assumption should rather be that users have no knowledge about the schema and the data.

This section discusses a class of approaches that aim to deal with this. The content can be summarized as follows:

- Motivated by the burden facing the data Web search users in specifying complex information needs, we identify a particular class of search paradigms that we refer to as schema-agnostic approaches.
- We conduct a systematic study of four widely used approaches representing the class of keyword-driven schema-agnostic search. In particular, we focus on IR-style keyword search, DB-style keyword search, keyword query translation and faceted search. All these approaches operate on an initial set of user keywords. We apply the process-oriented view of the Process-oriented Semantic Search model to investigate what conceptual steps these approaches require to address complex information needs.
- In the experiment, we carry out a task-based evaluation based on a large dataset and 19 participants. Based on the results, we derived the conclusions that keyword search and faceted search are effective and efficient for tasks which involve simple information needs. For complex needs, users proved to be more efficient with completion-based approaches and also, preferred them in terms of usability.

The keyword-driven schema-agnostic search paradigm is discussed in Section 5.2. The underlying process is presented in Section 5.2.2. Section 5.2.3 to Section 5.2.6 discuss the four types of approaches under consideration. Finally, the results of the experiments are discussed in Section 5.4.2.

### 5.2.1. Keyword-driven Schema-agnostic Search

Usable query interfaces have long been an active field of research. In this section, we identify and study search approaches, which do not require users to know the schema underlying the data. Work extensively studied
and also applied in practice that falls into this category includes IR-style keyword search, DB-style keyword search, natural language search, form-based search, faceted search and graphical query interfaces.

Clearly, keyword search is a popular search interface that does not require prior knowledge about the schema. As discussed for the motivating scenario, this is a simple and intuitive paradigm that is very effective for navigational search, i.e., to obtain some starting points. From these points, users however have to do further navigation and exploration to address complex information needs.

For specific domains and specific needs, experts redefine the types of complex queries that can be asked, and make them usable to lay users via customized forms. These form-based query interfaces have proven themselves effective for repetitive queries. They address common needs but fail on ad-hoc information needs. Another search paradigm that gains momentum is faceted search [DRM+08, RWD+08]. In fact, it can be seen as a special kind of form-based search. As opposed to a typical form-based interface, a faceted search interface is not restricted to the fields chosen by programmers but might contain any facets. Typically, facets are computed on-line for the given result set. Often, faceted search is also referred to as faceted browsing. This is because the user can use facets to explore and to iteratively refine the result sets.

Graphical query interfaces [Fog84] constitute a similar but less popular paradigm. Using such an interface, queries can be constructed using icons and visual elements that represent query constructs. One popular technique is to let users to construct the queries via drag and drop. Users can iteratively construct the query and modify the result set by drag-and-dropping visual query elements into a designated query box.

Natural language interfaces have been a research topic for decades and have found application in specialized domain such as expert systems [And95]. Recently, it has gained momentum has become also popular for open-domain settings. They are used by new and innovative Web search providers such as Powerset\(^1\) and True Knowledge\(^2\).

Besides NL interfaces, the database community has also been investigating the use of keywords for search [HWYY07, KPC+05, HGP03, LYMCO6, HP02, ACD02]. Unlike keyword search used on the Web, which focuses on simple needs, the keyword search elaborated here is used to obtain more complex results. Instead of a single set of resources,
the goal is to compute complex sets of resources and their relations. We refer to this work as result completion, as it can be implemented to suggest completions in the form of candidate results, given the user provided keywords [LJLF09]. As an alternative to computing candidate results, candidate interpretations in the form of structured queries can be computed [TWRC09]. Given the provided keywords, completions in the form of queries are presented from which the user chooses the intended one to obtain the results in a subsequent step. We refer to this work as query completion. These two paradigms can be regarded as extensions of auto-completion. However, auto-completion known from standard search interfaces operates at the level of words. Basically, a dictionary of terms is used to suggest candidate word completions as the user types. The completion approaches investigated here operate at a higher level, i.e., at the level of queries and results.

In this section, we focus on the study of schema-agnostic paradigms that are based on the use of keywords, i.e., keyword-driven schema-agnostic search. In particular, we will discuss IR-style keyword search, query completion, result completion as well as faceted search. We will now show that, from a process-oriented point of view, also faceted search relies on the use of keywords to obtain an initial result set and thus, falls into this category.

### 5.2.2. Keyword-driven Schema-agnostic Search Process

We will firstly investigate the search process. The goal is to identify the main conceptual steps each approach requires to address complex information needs. In Figs. (5.2.2+5.2.2), we illustrate these steps for the approaches under investigation. Common to the approaches are the phases (1) specifying needs using keywords, (2) inspecting initial results and (3) further browsing, analysis and retrieval of the final results. With keyword search, there is not much help from the system to retrieve complex results. The user needs to analyze and browse at the level of resources to collect the items of interest. Faceted search supports users in inspecting, analyzing and browsing at the level of facets. With completion-based approaches, users might retrieve complex answers right away – in the form of tuples that contain resources and their relations. We will now discuss state of the art techniques and the individual steps involved in each approach.
5.2.3. Keyword Search and Resource-based Browsing

5.2.3.1. The State of the Art

Keyword search is a paradigm commonly used by Web search engines to enable the retrieval of documents. Recently, a number of Semantic Web Search engines such as Hermes [TWH09], FalconS [CQ09] and Sindice [TDO07] have been developed, which also primarily rely on keyword search. Instead of documents, semantic entities (i.e., RDF resources) are returned. All these search systems are built upon the same basic concepts: (1) term-based representation of resources and queries (also called bag-of-words) and (2) term-based matching of queries against resources. Different matching and ranking techniques are employed. The ones frequently used are based on (an adopted version of) extensively studied information retrieval (IR) models such as the vector space model [WZW85] and the probabilistic models [RMC82]. The ranking schemes employed by these models leverage the discriminative quality of terms such as TFIDF and au-
thority of resources derived via PageRank [RD01]. Besides, also structure information might be used for ranking. In BM25F [RZT04] for instance, different weights are used to associate fields of documents (or generally speaking: properties of resources) with varying degrees of importance. This is to implement the intuition that certain properties are more discriminative and thus, more important than others, e.g. terms matching name shall yield higher scores than terms matching comment. Besides the class of probabilistic models such as Okapi BM25 and its extensions, current state of the art IR approaches have embraced the use of language modelling [PC98]. This approach assumes that resources and expressions of information needs are objects of the same type, and assesses their match by adopting techniques of language modeling from speech and natural language processing.

In the actual implementation, the bag-of-words representation of documents are stored in an inverted index (along with scores). While commercial solutions rely on their own infrastructures, Lucene has been widely used for the implementation of Semantic Web Search engines, Sindice, Hermes and FalconS in particular.

5.2.3.2. The Process

Using a keyword search system, the user starts by entering a list of keywords. The system matches this keyword query against resources to return a ranked list of results. This might contain the immediate items of interest, which is often the case with entity and fact search. However, most keyword search performed on the Web is navigational, i.e., to obtain initial results that are then used as the starting point for further browsing. In particular for relation search (i.e., for retrieving complex information), it is necessary that the user starts with navigational search. Then, the user inspects the result set, chooses the relevant resource(s) from which further navigation is performed to collect all the items of interest. The navigation is resource-based, i.e., is performed by following links between resources.

Example 11 Mary enters researchers AIFB to obtain the corresponding list of researchers. For every researcher, Mary follows the links to research work such as publications.
5.2.4. Keyword Search and Facet-based Browsing & Search

5.2.4.1. The State of the art

Browsing through the complex space of Web resources (e.g. documents, data) is a difficult task. The user might lose orientation along the way. Faceted search has become a popular paradigm as it helps to address this problem by providing facets for the user to inspect and navigate through the resources. Basically, facets correspond to properties, i.e., attributes of the underlying resources or relations between them. Faceted search is widely used in commercial systems such as on-line shops. In that setting, faceted search is conceptually similar to form-based search. The programmers predefine a fixed set of facets based on which the user can define (refine) the items of interest in terms of specific facets and facet values. Recently, advanced faceted search systems have been developed to search generic sets of resources, such as Freebase Parallax\(^3\) for dealing with domain-independent datasets. Based on the resource schema, the system automatically computes facets for a given set of resources. Faceted search of this kind has been proposed for searching documents [HSLY03, DRM\(^+\)08], for databases [DIW05, RWD\(^+\)08], as well as for RDF data [SWRS06, HMS\(^+\)05].

Currently, research in this area is concerned with the aspects of efficiency and effectiveness. For instance, an efficient implementation of faceted search has been proposed based on the inverted index [BYGH\(^+\)08]. The Lucene index is extended to store not only terms but also the facets to which they belong. Since the number of facets that can be derived for a result set might be large, the ranking of facets has attracted interest. Widely used in faceted search systems is frequency-based ranking, which is based on the count of values that are associated with a facet [DIW05]. Based on the similar idea, set-cover ranking has been suggested to maximize the number of distinct objects that are accessible from the top-k ranked facets [DIW05]. A more elaborated metric is proposed by Dash et al. to incorporate the notion of interestingness [DRM\(^+\)08]. Basically, it ranks those facets high which lead to surprising items, given a certain expectation. A different direction is to minimize the cost for the user finding some items of interest, where cost is derived from the facet hierarchy [RWD\(^+\)08], or more specifically, the operations the user performed on it [DIW05].

\(^3\)www.freebase.com/labs/parallax/
5.2.4.2. The Process

In order to retrieve complex result sets, the user browses the set of resources obtained from keyword search. However, instead of operating at the level of resources, the user firstly obtains an overview of the resources in the form of facets. The user inspects the facets and then, navigates along these high-level resource descriptions. By adding or removing facets, the user refines and expands the current result set. This is performed until finding and collecting all relevant items of interest. Since facet search should be regarded as an additional feature, the navigation at the resource level is still possible, and might be required in some cases.

**Example 12** After obtaining the list of resources for researchers AIFB, the user inspects the facets describing these results. This includes name, address, affiliation as well as research work. The last facet can be further decomposed into publication and project. The user navigates along these facets, and chooses to add the facet publication. The resulting list of resources now contains all publications from researchers at AIFB.

5.2.5. Keyword Search and Result Completion

5.2.5.1. The State of the art

With databases, the standard use case is to obtain complex results in the form of tuples instead of documents. Also here, keyword search is recognized as an intuitive paradigm that is more assessable to lay users. The underlying technique to support retrieval of tuples using keywords leverages some concepts widely used in the IR community. In fact, IR-style ranking schemes based on TFIDF as well as an adopted version of PageRank have been applied [HGP03, LYMC06, TWRC09]. Relations are represented using the bag-of-words model and in some systems, are also stored in an inverted index [TWRC09]. However, keywords are not matched against documents but against values (i.e., treated as terms) located in columns that are part of some database relations. More importantly, keywords of a single query might match values of a number of tuples that might be located in several relations. Thus, the matching technique employed here goes beyond simple IR-style matching. It involves additional processing to obtain sets of connected tuples, which contain all the query keywords. In particular, tuples containing some of the query keywords have to be joined such that the task amounts to finding join graphs of tuples that contain all the keywords. Each join graph is a complex tuple
(that might combine results from different relations), representing the answer to the keyword query. State of the art techniques in this area can be distinguished into two classes, i.e., (1) schema-based approaches and (2) schema-agnostic approaches that directly operate at the data level. Example systems of the first type are mostly implemented as database extensions, such as DBXplorer [ACD02] and Discover [HP02]. These systems translate keywords to candidate networks, which are essentially join expressions constructed using information given in the schema. These networks are then used to instantiate a number of SQL queries, which are finally executed using the underlying query engines. The results are returned as answers to the keyword query. The schema-agnostic approaches use customized indexes to match query terms against data tuples. As opposed to the schema-based approaches, the data that has to be explored for join graphs of tuples is potentially large. Thus, building the appropriate indexes to leverage materialized paths is one direction of addressing the efficiency issue [HWYY07]. Another main direction of research is to investigate efficient procedures for top-k join graph [TWRC09] (or Steiner-trees [KPC+05]) computation. Instead of returning all the results after the user finished typing the entire query, Li et al. [LJLF09] suggest to return top ranked candidate answers as completions for the keywords the user has finished typing. Thus, we refer to this kind of approaches as keyword-driven result completion.

5.2.5.2. The Process

Instead of a list of resources, the user obtains complex results for the entered keywords in the form of tuples. In particular, the user obtains candidate results upon entering every keyword of the query. The user inspects the result tuples. When the system correctly interprets the information need as expressed in terms of keywords, the process finishes after the user has entered the entire query. The need has been satisfied right away such that no additional effort is needed for browsing and collecting results. Otherwise, the user might choose some resources in the result set, and continues with resource-based navigation and retrieval of additional results, just like with standard keyword search.

Example 13  Upon the user entering every keyword of the query publication researchers AIFB, the system suggests different candidate results, i.e., top ranked list of publications after the first keyword publication has been entered, then all publications and associated researchers after the
user finished adding researchers to the query and finally, after the user added AIFB, top ranked tuples containing publications of researchers from AIFB are presented. The process stops here as the need has been satisfied.

5.2.6. Keyword Search and Query Completion

5.2.6.1. The State of the art

Recognizing that returning queries instead of answers might improve the type of expressive keyword search discussed previously, we proposed an approach referred to as keyword-driven query completion [TWRC09]. The queries computed represent possible interpretations of the keywords. The idea is to present a list of candidate interpretations, from which the user can choose the intended one. The main advantages of this approach over result completion are that (1) queries can be seen as descriptions that might facilitate users in inspecting and understanding the results, (2) they enable more effective refinement (compared to operating at the level of results) and also, (3) irrelevant results can be reduced because the system computes only answers to the intended query. The technique behind this approach is similar to the one used by schema-based result completion [ACD02, HP02] discussed previously. Schema information is used to search for join expressions, which can meaningfully connect elements that match the keywords. These join expressions are finally mapped to queries, representing possible interpretations of the keywords.

5.2.6.2. The Process

Upon the user entering a keyword, the system suggests queries that represent interpretations of the keywords entered so far. The user continues typing keywords and inspecting queries, until finding a query completion that matches the intended meaning. The user poses this query against the engine to obtain complex result tuples. In cases the computed interpretations do not perfectly match the intended meaning, the user might choose to inspect specific resources and to navigate along their links, just like with standard keyword search.

Example 14 After entering research work, the system suggests (among others), a query that retrieves all research work. As the user finishes typing researchers, the system suggests a query that retrieves all research work along with the associated researchers. Finally, after the entire query
research work researchers AIFB has been finished, one query returned by the system represents the intended meaning, i.e., to retrieve all research work from researchers at AIFB. In this example, the process also stops here.

5.3. Schema-agnostic Query Construction in SemSearchPro

Much work has been carried out on keyword search over relational data, tree-structured XML data (e.g. [ACD02], [CMKS03], [GSBS03], [HP02], [HGP03], [HHP06], [LYMC06], [KS06]) as well as graph-structured data (e.g. [HWYY07], [KPC+05], [BHN+02]) such as resources in SemSearchPro. The basic idea is to map keywords to data elements (keyword elements), search for substructures on the data graph that connect the keyword elements, and output the top-$k$ substructures computed on the basis of a scoring function. This task can be decomposed into (1) keyword mapping, (2) graph exploration, (3) scoring and (4) top-$k$ computation.

In existing approaches ([BHN+02, KPC+05]), an exact matching between keywords and labels of data elements is performed to obtain the keyword elements. For the exploration of the data graph, the so-called distinct root assumption is employed (see [HWYY07, BHN+02, KPC+05]). Under this assumption, only substructures in the form of trees with distinct roots are computed and the root element is assumed to be the answer. The algorithms for finding these answers also called Steiner trees are backward search ([BHN+02]) and bidirectional search ([KPC+05]). Using spread activation to choose also forward edges, the latter proves to be more efficient than the former, which traverses only backward edges to find the common root of the keyword elements of a given query. For top-$k$ processing and ranking, different scoring functions have been proposed (see [ACD02, HP02, GSBS03, HHP06, GSW05, LYMC06, AYKM+05]), where metrics range from path lengths to more complex measures adopted from IR. In order to guarantee that the computed answers indeed have the best scores, both the lower bound of the computed substructures and the upper bound of the remaining candidates have to be maintained. Since book-keeping this information is difficult and expensive, current algorithms compute the best answers only in an approximate way (see [BHN+02, KPC+05]).

In this section, we present the SemSearchPro implementation of the translation framework. It advances the state of the art in the following aspects:
• **Imprecise Syntactic and Semantic Matching** In SemSearchPro, IR concepts are adopted to support an imprecise matching of keywords against data elements. Data elements are modeled as documents and their labels are treated as document terms. The matching of keywords against these terms is evaluated using different syntactic similarity measures. Also, we have enriched every document with semantically similar terms extracted from Wordnet⁴ (a lexical database). This way, we incorporate both syntactic and semantic similarities.

• **Keyword Search through Query Translation** While the mentioned approaches interpret keywords as elements of answers, we interpret keywords as elements of structured queries. Instead of presenting the top-$k$ answers, which might actually belong to many distinct queries, we let the user select one of the top-$k$ queries to retrieve all its answers. This additional step is complementary since algorithms elaborated in previous approaches for top-$k$ answer computation (e.g. [HGP03],[ACD02],[HP02]) can still be applied after queries have been computed. Also, it might be beneficial to the user because queries can be seen as descriptions, and can thus facilitate the comprehension of the answers. Another advantage of this is that refinement can be made more precisely on the structured query than on the keyword query.

• **Algorithms for Subgraph Exploration** Our main technical contribution is a novel algorithm for the computation of the top-$k$ subgraphs. In current approaches, keywords are exclusively mapped to vertices. In order to connect the vertices corresponding to the keywords, current algorithms aim at computing tree-shaped candidate networks (e.g. [HGP03, ACD02, HP02]) or answer trees ([BHN⁺02, KPC⁺05, HWYY07]). Since keywords do not necessarily correspond to answers exclusively in our approach, they might also be mapped to edges. As a consequence, substructures connecting keyword elements are not restricted to trees, but can be graphs in general. Thus, algorithms as applied for tree-exploration such as breadth-first search (e.g. [ACD02], [HP02]), backward search [BHN⁺02] or bidirectional search [KPC⁺05] are not sufficient.

• **Efficient and Complete Top-$k$ through Graph Summarization** So far, algorithms for top-$k$ retrieval assume that the computed sub-
structures connecting keyword elements represent trees with distinct roots (e.g. [BHN+02], [HWYY07], [KPC+05]). Since the book-keeping of information required for top-$k$ processing is difficult and expensive, existing top-$k$ algorithms [KPC+05, BHN+02] can not provide the guarantee that the results indeed have the best scores. This problem is exacerbated when searching for subgraphs. In order to guarantee that the results are indeed top-$k$ subgraphs, we introduce more complex data structures to keep track of the scores of all explored paths and of all remaining candidates. For efficiency reasons, a strategy for graph summarization is employed that can substantially reduce the search space. By graph summarization, we basically mean the clustering technique we proposed for computing a pseudo-schema from the data graph as discussed in Section 4.3.2 of Chapter 4. This pseudo-schema is enhanced with keyword elements that can be found for a given query, to construct a query space. The exploration of subgraphs does not operate on the entire data graph but on this much reduced query space containing only the elements that are necessary to compute the queries.

We have achieved encouraging performance in comparison with previous approaches [KPC+05, HWYY07]. The effectiveness studies show that the generated queries match the meaning intended by the user very well. The user feedbacks on the demo system (i.e., the Hermes system as presented in Section 4.4 in Chapter 4) suggest that the presentation of structured queries is valuable in terms of comprehension and enables more precise refinement than using keywords.

We will begin with an overview in Section 5.3.1 to introduce the problem and briefly sketch the SemSearchPro solution in terms of the offline tasks and online computation that are required. Details on the computation of the indexes, the scores and the translation are then provided in Section 5.3.2, Section 5.3.3 and Section 5.3.4, respectively. A comparison with related work is provided in Section 5.3.5. Finally, the evaluation results are discussed in Section 5.4.1.

### 5.3.1. Overview of the Approach

This approach implements the semantic translation framework of the process-oriented semantic search model. User queries $Q_U$ are simply sets of keywords $(k_1, ..., k_i)$ and system queries $Q_S$ are of the type conjunctive queries. Both the translation of $Q_U$ to $Q_S$ and the actual processing of
Figure 5.4. Example resource model.

$Q_s$ make use of the underlying SemSearchPro’s graph-structured resource model $\mathcal{R}$ (also called data graph).

5.3.1.1. The Translation Problem

Thus, the specific translation problem considered here is the computation of conjunctive queries from keywords using the graph-structured resource model. In particular, the goal is to find the top-ranked queries, where the ranking is produced by the application of a cost function $C : q \mapsto c$. For any computed system query $Q$, $C$ assigns a cost that captures the degree of relevance of $Q$, i.e., the degree to which the answers that can be obtained by processing $Q$ satisfy the user information need. In particular, for any two queries $Q_1$ and $Q_2$, $C(Q_1)$ should be lower than or equal to $C(Q_2)$ if given the information need, the answers that can be obtained with $Q_1$ are more or as relevant as the answers obtained with $Q_2$.

Example 15 As a running example, Fig. 5.4 shows an example data graph containing data about publications, researchers, the institutes they work for etc. Technically, a graph structured resource model such as an RDF data graph can be managed using customized solutions (e.g., graph databases, RDF stores) or simply be stored in a relational database. For instance, exactly one relational table of three columns can be used to store entities’ properties and attributes. In this case, the data graph can be mapped to the table shown in Fig. 5.5(a). The example query asked by the user is shown in Fig. 5.5(b). The user intention is to retrieve things...
Chapter 5. Query Construction and Refinement

Example Keyword Query
2006 cimiano aifb

Example Conjunctive Query
\((x, y, z). type(x, \text{Publication}) \land \text{year}(x, 2006) \land \text{author}(x, y) \land \text{name}(y, 'P. Cimiano') \land \text{worksAt}(y, z) \land \text{name}(z, 'AIFB')\)

Example SPARQL Query
```
SELECT ?x, ?y, ?z WHERE {
  ?x type \text{Publication}. ?x \text{year} 2006.
  ?x \text{author} ?y. ?y \text{name} 'P. Cimiano'.
  ?y \text{worksAt} ?z. ?z \text{name} 'AIFB'}
```

Example SQL Query
```
SELECT A.p, D.s, F.s.
FROM AS xA AS, E xA SB AS, E xA SC AS,
EX AS D AS, EX AS E AS, EX AS F AS
WHERE A.p = \text{type}
  AND A.o = \text{Publication} AND A.s = \text{B.s}.
  AND B.p = \text{year} AND B.o = '2006'
  AND B.s = C.s AND C.p = \text{author}
  AND C.o = D.s AND D.p = \text{name}
  AND D.s = E.s AND D.o = 'P. Cimiano'
  AND E.p = \text{worksAt} AND E.o = F.s.
  AND F.p = \text{name} AND F.o = 'AIFB'
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pro2URI</td>
<td>type</td>
<td>Project</td>
</tr>
<tr>
<td>pro1URI</td>
<td>type</td>
<td>Project</td>
</tr>
<tr>
<td>pro1URI</td>
<td>name</td>
<td>X-Media</td>
</tr>
<tr>
<td>pub1URI</td>
<td>type</td>
<td>Publication</td>
</tr>
<tr>
<td>pub1URI</td>
<td>author</td>
<td>re1URI</td>
</tr>
<tr>
<td>pub1URI</td>
<td>author</td>
<td>re2URI</td>
</tr>
<tr>
<td>pub1URI</td>
<td>year</td>
<td>2006</td>
</tr>
<tr>
<td>pub2URI</td>
<td>type</td>
<td>Publication</td>
</tr>
<tr>
<td>re1URI</td>
<td>type</td>
<td>Researcher</td>
</tr>
<tr>
<td>re1URI</td>
<td>worksAt</td>
<td>inst1URI</td>
</tr>
<tr>
<td>re2URI</td>
<td>type</td>
<td>Researcher</td>
</tr>
<tr>
<td>re2URI</td>
<td>name</td>
<td>Philipp</td>
</tr>
<tr>
<td>inst1URI</td>
<td>type</td>
<td>Institute</td>
</tr>
<tr>
<td>inst1URI</td>
<td>name</td>
<td>AIFB</td>
</tr>
<tr>
<td>inst2URI</td>
<td>type</td>
<td>Institute</td>
</tr>
</tbody>
</table>

Figure 5.5. (a) Single table schema and (b) example queries.

in 2006 from a researcher with the name cimiano who works at AIFB. The conjunctive query which matches this user need is shown below the example keyword query. To illustrate the correspondence of conjunctive queries to widely used query languages, two equivalent queries written in SQL and SPARQL are also given. The SQL query selects information from the table in Fig. 5.5(a).

The challenge here is to infer a structured query from a “limited” representation of the information need, i.e., from a set keywords. One contribution towards addressing this challenge is to use the semantics in the data to accomplish this. An extensive report on the rationales and technical details behind this idea can be found in our conference publication [TCRS07]. In terms of the semantics given in the data, the user intention might be interpreted as \text{Publication}s that are from the \text{year} 2006 written by the \text{author} with the name Philipp Cimiano who \text{worksAt} AIFB. However, the keyword query has no reference to many of those elements in the data, e.g., \text{Publication}, \text{author}, \text{year}, \text{worksAt}. Thus, these elements and connections between them need to be inferred to translate the query correctly. We have discussed in [WZL+08] that a pseudo-schema constructed from the data can be used to infer missing connections. In [TWRC09], we were concerned with the challenge of doing this effi-
ciently and propose an approach in which the best interpretations of the query are computed using a top-$k$ algorithm. We will now discuss the main ideas behind these approaches.

5.3.1.2. Overview of the Solution

We start with an overview of the different steps involved in the process of keyword search, which is depicted in Fig. 5.6. We will illustrate the whole process on the basis of a running example for the query ‘X-Media Philipp Cimiano publications’.

In order to compute queries from keywords, the keywords are first mapped to elements of the data graph. These keyword elements are combined with the semantic model to construct the query space. This is then explored to find a connecting element, i.e., a particular type of graph element that is connected to all keyword elements. The paths between the connecting element and a keyword element are combined to construct a matching subgraph. For each such subgraph, a conjunctive query is derived through the mapping of graph elements to query elements. In particular, based on the structural correspondence of triple patterns of a query and edges of the data graph, vertices are mapped to variables or constants, and edges are mapped to predicates. The process continues until the top-$k$ queries have been computed.

In order to perform these steps in an efficient way, we preprocess the data graph to obtain a keyword index that is used for the keyword-to-element mapping. For exploration, a semantic model index is constructed for fast access. This index stores elements of the semantic model that is computed using the clustering technique as discussed previously in Sec-
tion 4.3.2 of Chapter 4. Recall that this model is basically a summary of the original data graph. At the time of query processing, the semantic model index is augmented with keyword elements obtained from the keyword-to-element mapping to compute the query space. This space contains sufficient information to derive the structure as well as the predicates and constants of the query. Since we are interested in the top-$k$ queries, graph elements are also augmented with scores. While scores associated with structure elements can be computed off-line, scores of keyword elements are specific to the query and thus can only be processed at query computation time.

Example 16 In Fig. 5.7, the query space is shown for the data graph in Fig. 5.4. This query space contains both the structural information computed during preprocessing (rendered gray) and the query specific elements added at query computation time (shown in white). In particular, the keywords of the query given in Fig. 5.5(b) are mapped to corresponding elements of the query space, i.e., the vertices $v_{2008}$, $v_{AIFB}$ and $v_{Philipp Cimiano}$. For each of these keyword elements, the score as computed off-line is combined with the online matching score obtained from the keyword-to-element mapping. The graph exploration starts from these three vertices (corresponding to keywords in the query), resulting in three different paths as shown by the different arrow types in Fig. 5.7 (note that the arrows indicate the direction of the exploration, not the direction of the edges). Among them, there are three paths that meet at the connecting element $n_c$, namely $p_1(AIFB, ..., n_c)$, $p_2(Philipp Cimiano, ..., n_c)$ and $p_2(2006, year, n_c)$ (rendered bold in Fig. 5.7). These three paths are merged, and the resulting matching subgraph is mapped to the conjunctive query presented in Fig. 5.5(b). Note that in this case, the matching subgraph is actually a tree. However, keyword elements might be edges that form a cyclic graph. The matching substructure is also very likely to be a graph when keywords are mapped to edges that form loops. We will see that the query space might contain many loops.

5.3.2. Offline Data Indexing

This section describes the off-line indexing process where the data graph is preprocessed and stored in specific data structures of a keyword and a semantic model index. Both indexes are employed to support the computation of queries. The former is used for the mapping of keywords to
elements of the graph and the latter is for graph exploration.

5.3.2.1. The Keyword Index

In SemSearchPro, keywords entered by the user might refer to data elements (constants) or structure elements of a query (predicates). From the data graph point of view, keywords might refer to C-vertices (classes), E-vertices (entities) or V-vertices (data values) and edges. However, we choose to omit E-vertices in the indexing process as it can be assumed the user will enter keywords corresponding to attribute values such as a name rather than using the verbose identifier of an E-vertex to refer to the entity in question. Besides, keywords might stand for structure elements, i.e., refer to labels of edges. Thus, our approach differs from other state of the art approaches in that keywords can also refer to relations (edges) between nodes. This is important and makes clear the distinction between computing queries and computing answers. For computing queries, it is essential to assume that keywords might correspond to edge labels, from which query predicates can be derived.

Conceptually, the keyword index is a keyword-element map. It is used for the evaluation of a multi-valued function \( f : K \mapsto 2^{V_C \cup V_E \cup V_A \cup E} \), which for each keyword \( k_i \in K \), returns the set of corresponding graph elements \( N_i \) (called keyword elements). In the special cases where the keyword corresponds to a V-vertex or an A-edge, more complex data structures are required. In particular,
for a $V$-vertex, a data structure of the form $[V$-vertex, $A$-edge, $(C$-vertex$_1$, ..., $C$-vertex$_n)]$ is returned. Elements stored in this data structure are neighbors of the $V$-vertex, namely those connected through the edges \text{type}(E$-vertex, $C$-vertex$_j)$ where $1 \leq j \leq n$ and $A$-edge$(E$-vertex, $V$-vertex).

Likewise, for an $A$-edge$(E$-vertex, $V$-vertex), the data structure $[A$-edge, $(C$-vertex$_1$, ..., $C$-vertex$_n)]$ is used to return also the neighbor’s class memberships. Using these data structures, query-specific keyword elements can be added to the semantic model to construct the query space on-the-fly.

In order to recognize also keywords that do not exactly match labels of data elements, the keyword-element map is implemented as an inverted index. In particular, a lexical analysis (stemming, removal of stopwords) as supported by standard IR engines (c.f. Lucene\textsuperscript{5}) is performed on the labels of elements in $V^S_S \cup V^R_V \cup E^R_R$ in order to obtain terms. Processing labels consisting of more than one word might result in many terms. Then, a list of references to the corresponding graph elements is created for every term. Further, semantically similar entries such as synonyms, hyponyms and hypernyms are extracted from WordNet for every term. Every such entry is linked with the list of references of the respective term. Thus, graph elements can be returned also in the cases where the given keyword is semantically related with a term extracted from the elements’ labels. In order to incorporate syntactic similarities, the Levenshtein distance is used for an imprecise matching of keywords to terms.

Thus, the keyword-element map is in fact an IR engine, which lexically analyzes a given keyword, performs an imprecise matching, and finally, returns a list of graph elements having labels that are syntactically or semantically similar.

5.3.2.2. The Semantic Model Index

In current approaches, the underlying model used for keyword search is simply the entire data graph [BHN\textsuperscript{+}02, KPC\textsuperscript{+}05, HWYY07]. This might be very expensive when the data graph is large. As opposed to these approaches which focus on computing answers, SemSearchPro is concerned with computing queries. For this purpose, is suffices to explore the semantic model instead of the data graph. The semantic model used here is in fact a pseudo-schema. For more details on this model and the clustering

\textsuperscript{5}http://lucene.apache.org
technique used to compute it, please refer to Section 4.3.2 of Chapter 4.

Elements of the semantic model are stored in a particular index called the semantic model index. Since the semantic model is essentially a graph, the same solution used for indexing and managing graph-structured data (RDF) can be employed here. We employ Semplore, an RDF store that makes use of an underlying IR engine and the inverted index infrastructure it provides. More details on Semplore and RDF storage and indexing can be found in Section 3.4 in Chapter 3.

For storing and accessing elements of the semantic model, we use the inverted indexes employed by Semplore. Whenever the semantic model is small enough, it is fetched from disk beforehand and loaded into memory.

### 5.3.3. Offline Data Scoring

The translation process can result in many queries all corresponding to possible interpretations of the keywords. This section introduces several scoring functions that aim to assess the relevance of the computed queries.

The scoring of answers has been extensively discussed in the database and IR communities (see [GSW05, HGP03, HHP06, LYMC06, GSBS03, AYKM+05]). In the context of graph-structured data, metrics proposed often incorporate both the graph structure and the label of graph elements. Standard metrics that can be computed off-line are PageRank (for scoring vertices) and shortest distance (for scoring paths). A widely used metric that is computed on-the-fly for a given query is TF/IDF (for scoring keyword elements).

Instead of answer trees (see [HWYY07, BHN+02, KPC+05]), we consider the subgraphs \( q \in Q \) from which queries will be derived. These graphs are constructed from a set of paths \( P(q) \). The score of such a graph is defined as a monotonic aggregation of its paths’ scores. In particular,

\[
C(q) = \sum_{p_i \in P(q)} C(p_i)
\] (5.1)

is used, where \( C(p_i) \) and \( C(q) \) are in fact not scores, but denote the costs. In general, the cost of a path is computed from the cost of its elements, i.e.,

\[
C(p_i) = \sum_{n \in p_i} C(n)
\] (5.2)

We will now discuss the metrics we have adopted to obtain different schemes for the computation of the path’s cost.
5.3.3.1. Path Length

The path length is commonly used as a basic metric for ranking answer trees in recent approaches to keyword queries (e.g. [KPC+05], [HWYY07]). Generally, this is based on the assumption that the information need of the user can be modelled in terms of entities, which are closely related [TCRS07]. Thus, a shorter path between two entities (keyword elements) should be preferred. For instance, given the keywords “X-media” and “Cimiano”, the path along the graph elements (Philipp Cimiano, worksAt, X-Media) more likely corresponds to the user’s intention than (Philipp Cimiano, author, hasProject, X-Media). For computing path length, the general cost function for paths given above can be casted as

\[ C(p_i) = \sum_{n \in p_i} 1 \]  

(5.3)

I.e., the cost of an element is simply one. Accordingly, the score of a graph can be computed via

\[ C_1(q) = \sum_{p_i \in P} \sum_{n \in p_i} 1 \]  

(5.4)

5.3.3.2. Popularity Score

The previous function can be further extended to exploit structure information in the graph. For this, we use

\[ C_2(q) = \sum_{p_i \in P} \sum_{n \in p_i} c(n) \]  

(5.5)

where \( c(n) \) is an element-specific cost function. In particular, for a given semantic model \( S(V^S_C, L_R, E^S) \), we define

- \( c(v^S_C) = 1 - \frac{|v^R_E|}{|V^S_C|} \) for class vertices \( v^S_C \), where \( |V^S_C| \) is the total number of vertices in the semantic model and \( |v^R_E| \) is the number of E-vertices that have been clustered to the C-vertex \( v^S_C \) during the construction of \( S \),

- \( c(e(v^S_{C_i}, v^S_{C_j})) = 1 - \frac{|e(v^R_{E_i}, v^R_{E_j})|}{|E^S|} \) for relation edges \( e(v^S_{C_i}, v^S_{C_j}) \), where \( |E^S| \) is the total number of edges and \( |e(v^R_{E_i}, v^R_{E_j})| \) is the number of data graph edges that have been clustered to the corresponding edge \( e(v^S_{C_i}, v^S_{C_j}) \) of \( S \).
Since both local frequency (w.r.t. an aggregated element) and the total number of occurrences are considered, the computed metric bears resemblance to the notion of TFIDF, a metric commonly used in IR for ranking documents. These cost functions aim to capture the “popularity” of an element of the semantic model, measured by the relative number of data elements that it actually represents. The higher the popularity, the lower should its contribution be to the cost of a path.

5.3.3.3. Keyword Matching Score

A special treatment of the keyword matching elements can be achieved through

$$C_3(q) = \sum_{p_i \in P} \sum_{n \in p_i} c(n) s_m(n)$$

(5.6)

where $s_m(n)$ is the matching score of an element $n$. This matching score is $[0, 1]$ if $n$ is a keyword element (correspond to a keyword), otherwise it is simply set to 1.

The higher $s_m(n)$, the lower should be the contribution of a keyword element to the cost of a path. In our approach, $s_m(n)$ reflects both syntactic and semantic similarity (by incorporating knowledge from a lexical resource such as WordNet) between keywords and labels of graph elements.

While the path lengths and the popularity scores can be computed offline, the matching scores are query specific and are thus computed and associated with elements of the semantic model during query computation. Also, note that the costs of individual paths are computed independently, such that if the paths share the same element, the cost of this element will be counted multiple times. As noted in [HWYY07], this has the advantage that the preferred graphs exhibit tighter connections between keyword elements. This is in line with the assumption that closely connected entities more likely match the users’ information need [TCRS07]. We will show later that this also facilitates top-$k$ processing due to the fact that the cost of paths can be computed “locally”.

5.3.4. Online Query Translation

For query computation, five tasks need to be performed: (1) mapping of keywords to data elements to obtain keyword elements, (2) computation
of the query space, (3) exploration of the query space to find matching subgraphs connecting the keyword elements, (4) top-\(k\) processing and (5) generation of the query for the top-\(k\) matching subgraphs.

5.3.4.1. Mapping Keywords to Keyword Elements

For the first task, the keyword index is used to obtain a possibly overlapping set \(K_i\) of graph elements for every given keyword. Thus, every element in this set might be representative for one or several keywords.

For computing \(K_i\), every keyword of the query is submitted against the IR engine that manages the keyword index. The result is a list of matching elements along with scores that reflect the IR-style relevance. For reasons of efficiency, a threshold can be set here to filter out a possibly large amount of matching elements that have low scores.

5.3.4.2. Construction of the Query Space

The query space is a combination of the semantic model and the query specific keyword elements. Recall that attribute edges and value vertices of the resource model have not been considered in the construction of the semantic model. By definition, a value vertex has only one edge, namely an incoming attribute edge that connects it with an entity vertex. This means that in the exploration for matching subgraphs, any traversal along an attribute edge ends at a value vertex. Thus, both attribute edges and value vertices do not help to connect keyword elements. They are relevant for query translation only when they are keyword elements themselves. Thus, in order to keep the search space minimal, the semantic model is augmented only with the relevant attribute edges and value vertices that are keyword elements:

**Definition 15** Given the user query \(q\) being the set of keywords \(K\) and the corresponding set of keyword matching elements \(N_K\), the query space \(S_Q(S, N_K)\) of a resource model \(R(V^R, L, E^R)\) consists of \(R\)’s semantic model \(S(V^S, L, E^S)\) additionally containing

- the edge \(e(v^S_C, v_k)\) iff there is a keyword matching vertex \(v_k \in N_K\), and \(type(v^R_E, v^S_C), e(v^R_E, v_k) \in E^R\),
- and the edge \(e_k(v^S_C, Value)\) iff there is a keyword matching edge \(e_k(v^R_E, \tilde{v}^R_E) \in N_K\) and \(type(v^R_E, v^S_C), e_k(v^R_E, \tilde{v}^R_E) \in E^R\) and \(\tilde{v}^R_E\) is not a keyword element. Value is simply an artificial node created to fill the target position of that edge.
Intuitively speaking, if the keyword element is value vertex \( v_K \), it is connected to a class in the semantic model, which contains an entity that has \( v_K \) as attribute value. If the keyword element is attribute edge with the label \( e_k \), it is considered only if its target vertex is not already a keyword element. Because in this case, it would already be added to the semantic model due to the first rule. If this is not the case, it will be added to the semantic model by means of a newly created edge with \( e_k \) as the label and \( Value \) as the “dummy” target vertex.

In order to construct \( S_Q \), we make use of the data structures resulting from the mapping, namely \([V\text{-vertex}, A\text{-edge}, (C\text{-vertex}_1, \ldots, C\text{-vertex}_n)]\) and \((A\text{-edge}, C\text{-vertex})\). Using neighbor elements given in this data, edges of the form \( A\text{-edge}(C\text{-vertex}_i, V\text{-vertex}) \) are added to \( S \) for every keyword matching \( V\text{-vertex} \), and an edge of the form \( A\text{-edge}(C\text{-vertex}, Value) \) is added for every keyword matching \( A\text{-edge} \).

Note that only \( A\text{-edges} \) and \( V\text{-vertices} \) are added as \( S \) already contains the \( C\text{-vertices} \).

### 5.3.4.3. Algorithms for Computing Matching Subgraphs

Given the keyword elements, the objective of the exploration is to find minimal substructures in the graph that connect these elements. In particular, we search for minimal subgraphs that include one representative of every keyword. This notion of a minimal matching subgraph is formalized as follows.

**Definition 16** Let \( R(V^R, L, E^R) \) be the graph-structured resource model, \( K = \{k_1, \ldots, k_n\} \) be a set of keywords and let \( f : K \mapsto 2^{V^R \cup E^R} \) be a function that maps keywords to sets of corresponding graph elements. A \( K \)-matching subgraph of \( R \) is a graph \( q(V^Q, L^Q, E^Q) \) with \( V^Q \subseteq V^R \), \( L^Q \subseteq L \) and \( E^Q \subseteq E^R \) such that

- for every \( k \in K \), \( f(k) \cap (V^Q \cup E^Q) \neq \emptyset \), i.e., \( q \) contains at least one representative for every keyword from \( K \), and
- \( q \) is connected such that from every graph element to every other graph element from \( q \), there exists a path.

A matching graph \( q_i \) is minimal if there exist no other \( q_j \) such that \( C(q_j) < C(q_i) \).

Many approaches to keyword search on graph data or databases have dealt with the same problem, i.e., the one of finding substructures that
connect keywords. Approaches that operate on XML data rely on the exploration of tree-structured data (see [FKM00], [CMKS03], [GSBS03], [LYJ04]). More related to our work are approaches that deal with algorithms on graphs. We will now discuss and compare them with our approach.

**Backward Search** The backward search ([BHN+02]) algorithm starts from the keyword elements and then performs an iterative traversal along incoming edges of visited elements until finding a connecting element, called *answer root*. At each iteration, the element that is chosen for traversal is the one that has the shortest distance to the starting element.

**Bidirectional Search** Noticing that their backward search exhibits poor performance on certain graphs, the authors propose a bidirectional search algorithm (see [KPC+05]). The intuition is that from some vertices the answer root can be reached faster by following outgoing rather than incoming edges. For prioritization, heuristic activation factors are used in order to estimate how likely an edge will lead to an answer root. These factors are derived from the general graph topology and the elements that have been explored. While this search strategy has been shown to have good performance w.r.t. different graphs, there is no worst-case performance guarantee.

**Searching with Materialized Paths** Recently, an extension to the bidirectional search algorithms has been proposed, which provides such a guarantee, i.e., it has been proven to be \(m\)-optimal, where \(m\) is the number of keywords [HWYY07]. This basically means that in worst case, the vertices visited by the algorithm is no more than \(m\)-times the number of vertices visited by an optimal “oracle” algorithm. This optimality can be ensured through the use of additional connectivity information that is stored in the index. At each iteration, this information allows to determine the elements that are connected to a given keyword element as well as the shortest distance to that keyword element, thereby offering guidance and enabling a more goal-directed exploration. Since in principle any graph element could be a keyword element, encoding all this information also largely increases space complexity [HWYY07]. In the single-level index in [HWYYY07], space complexity of the index comes to \(O(N^2)\) where \(N\) is the size of the graph. While the proposed second-level index is more efficient, it is still very large. It is based on the partitioning of the graph into blocks, resulting in overall index at least the size of \(O(\sum_{b \in B} N_b^2 + |B||P|)\), where \(b\) is a block, \(N_b\) is the block size and \(P\) are special vertices connecting blocks called portals.
Compared with these approaches, we tackle a more general problem. It has been assumed in previous work that the keywords correspond to leaf vertices and the answer is the root vertex of a tree. In our approach, a keyword can represent any element in the graph, including an edge. Thus, instead of trees, substructures that connect these keyword elements might be graphs. Also, since the connecting element is not assumed to be the root of a tree, forward search is equally important as backward search. The technique for indexing distance information as discussed above is orthogonal. However, minimality is defined in terms of cost in our approach. Costs come in two fashions: *query-independent* costs which can be computed off-line (such as the distance between graph elements) and *query-specific* costs have to be computed on-the-fly. Techniques for indexing and materialization [HWYY07] can be applied to query-independent costs only.

We will now elaborate on our approach for finding matching subgraphs that are minimal w.r.t. to both query-specific and query-independent costs.

### 5.3.4.4. Search for Minimal Matching Subgraphs

The technique we propose for searching minimal matching subgraphs is presented in Alg. 1.

**Input and Data Structures** Computing the matching subgraphs of $\mathcal{R}$ is not based on $\mathcal{R}$ but the query space $S_Q$. The input to the Alg. 1 comprises the elements of the query space $S_Q$ and particularly, the keyword elements $K = (K_1, ..., K_m)$ where each $K_i$ corresponds to the set of data elements associated to keyword $i$ (which have been retrieved using the keyword index). Further, $k$ is used to denote the number of queries to be computed. The maximum distance $d_{\text{max}}$ is provided to constrain the exploration to neighbors that are within a given vicinity. The *cursor* concept is employed for the exploration. In order to keep track of the visited paths, every cursor is represented as $c(n, k, p, d, w)$, where $n$ is the graph element just visited, $k$ is a keyword element representing the origin of the path captured by $c$ and $p$ is the parent cursor of $c$. Using this data structure, the path between $n$ and $k$ captured by $c$ can be computed through the recursive traversal of the parent cursors. Besides, the cost $w$ and the distance $d$ (the length) is stored for the path. In order to keep track of information related to a graph element $n$ and the different paths discovered for $n$ during the exploration, a data structure of the form $(w, (C_1, ..., C_m))$ is employed, where $w$ is the cost of $n$ as discussed in Section 5.3.3 and $C_i$
Algorithm 1: Search for Minimal Matching Subgraphs

Input: $k, d_{max}; S_Q; K = (K_1, ..., K_m)$.
Data: $c(n, k, p, d, w); LQ = (Q_1, ..., Q_m); n(w, (C_1, ..., C_m)); kCAN$.
Result: the top-$k$ queries $kTOP$

1 // add cursor for each keyword element to $Q_i \in LQ$
2 foreach $K_i \in K$ do
3     foreach $k \in K_i$ do
4         $Q_i.add(newCursor (k, k, \emptyset, 0, k.w))$;
5     end
6 end
7 while not all queues $Q_i \in LQ$ are empty do
8     $c \leftarrow \text{minCostCursor}(LQ)$;
9     $n \leftarrow c.n$;
10    if $c.d < d_{max}$ then
11        $n.addCursor(c)$;
12        // all neighbors except parent element of $c$
13        $Neighbors \leftarrow \text{neighbors}(n) \setminus (c.p).n$;
14        // no more neighbours!
15        if $Neighbors \neq \emptyset$ then
16            foreach $n \in Neighbors$ do
17                // cyclic path when $n$ already visited by $c$
18                if $n \notin \text{parents}(c)$ then
19                    // add new cursor to respective queue
20                    $Q_i.add(newCursor (n, c.k, c.n, c.d + 1, c.w + n.w))$;
21                end
22            end
23        end
24        $Q_i.pop(c)$;
25    end
26    $kTOP \leftarrow \text{Top-k}(n, kCAN, LQ, k, kTOP)$;
27 end
28 return $kTOP$;
is a sorted list of cursors representing paths from \( k_i \) to \( n \). For supporting top-\( k \), \( kCAN \) is used as a global variable to keep track of the candidate subgraphs computed during the exploration.

**Initialization and General Idea** Similar to backward search, the exploration starts with a set of keyword elements. For each of these, cursors with an empty path history are created for the keyword elements and placed into the respective queue \( Q_i \in LQ \) (line 4). During exploration, the “cheapest” cursor created so far is selected for further expansion (line 8). Every cursor expansion constitutes an exploration step, where new cursors are created for the neighbors of the element just visited by the current cursor (line 13-20). At every step of the exploration, top-\( k \) is invoked to check whether the element just visited is a connecting element, and whether it is safe to terminate the process (line 25). The top-\( k \) computation is discussed in more detail later. Here we continue with the discussion of the actual exploration.

**Graph Exploration** At each iteration, a cursor \( c \) with the lowest cost is taken from \( Q_i \in LQ \) (line 8). Since \( Q_i \) is sorted according to the cursors’ cost, only the top element of each \( Q_i \) has to be considered to determine \( c \). In case that the current distance \( c.d \) does not exceed the parameter \( d_{\text{max}} \) (line 10), \( c \) is first added to the corresponding list \( C_i \) of \( n \), the graph element associated with \( c \) (line 11). This is to mark that \( n \) is connected with \( c.k \) through the path represented by \( c \). During top-\( k \) processing, these paths are used to verify whether a given element \( n \) is a connecting element. Then, the algorithm continues to explore the neighborhood of \( n \), expanding the current cursor by creating new cursors for all neighbor elements of \( n \) (except the parent node \((c.p).n\) that we have just visited) and add them to the respective queues \( Q_i \) (line 20). The distance and the cost of these new cursors are computed on the basis of the current cursor \( c \) and the cost function as discussed in Section 5.3.3. Since this cost function allows the contribution of every path to be computed independently, the cost of a new cursor is simply \( c.w + n.w \), where \( n \) is the neighbor element for which the new cursor has been created. Note that compared with the mentioned search algorithms, prioritization in our approach is based on the cost of the cursor’s path. Also, \( n \) might be a vertex or an edge. Thus, neighbors might be any incoming and outgoing edges, or vertices.

**Computation of Distinct Paths** The goal of the iterative expansion of cursors is to explore all possible distinct paths, beginning from some keyword elements. During this exploration, a graph element might be
explored many times. However, a cursor \( c_i \) is only expanded to a child cursor \( c_j \) if the neighbor element for which \( c_j \) should be created is not the parent element just visited before, i.e., \( (c_i.p).n \neq c_j.n \) (line 18). This is to prevent backward exploration of the current path as captured by \( c_i \). Also, \( c_j.n \) should not be an element of the current path, i.e., it is not one of the parent elements already visited by \( c_i \) (line 13). Such a cursor would result in a cyclic expansion. Thus, this type of cursors as well as the cursors that have been completely expanded to neighbors, are finally removed from the queue (line 24).

**Termination** The exploration terminates when one of the following conditions is satisfied: (1) all possible distinct paths have been computed such that there are no further cursors in \( LQ \) (2) all paths of a given length \( d_{\text{max}} \) have been explored for all keyword elements and (3) the top-\( k \) queries have been computed (see next subsection).

We now prove via an inductive argument that during the exploration, cursors are created in the order of the costs of the paths they represent. In other words, no cheap paths are left out in the exploration. We will see that this proposition is essential for top-\( k \) with best scores guarantee.

**Proposition 1** Given any execution stage of Alg. 1 after the initialization, let \( w_{\text{min}} = \min_{c \in LQ} c.w \) be the minimal cost occurring among the cursors from \( LQ \). Then for any graph element \( n \) and any keyword element \( n_k \) holds the following: if there is a path from \( n \) to \( n_k \) with lower cost than \( w_{\text{min}} \) then a cursor \( c_k = (n, n_k, c_{\text{parent}}, d, w) \) has already been created.

**Proof** This can be proven by induction on the length of the considered path from \( n \) to \( n_k \). As a base case, we have a path length of zero, i.e., \( n = n_k \). This claim is satisfied for this base case because corresponding cursors are created during the initialization.

For the induction step, consider some path \( p \) with nonzero length and suppose the contrary, i.e., the path \( p \) from \( n \) to \( n_k \) has cost \( w \) smaller than \( w_{\text{min}} \) but there is no cursor for \( n \) indicating this (claim †). Now let \( n' \) be the graph element next to \( n \) on this path. Then the subpath \( p' \) of \( p \) leading from \( n' \) to \( n_k \) has smaller length, whence we can apply the induction hypothesis concluding that there is a cursor \( c' = (n', n_k, c'_{\text{parent}}, d', w') \) assigned to \( n' \). As every path has higher cost than any of its subpaths, we know that \( w' < w \) and therefore \( w' < w_{\text{min}} \) due to assumption. Yet since \( w_{\text{min}} \) is the lowest cost occurring among all cursors from \( LQ \), \( c' \) must have been processed before and hence created a cursor for \( n \) as well. This contradicts our assumption that there is no such cursor for \( n \) (claim †) and thus proves
the induction step.

5.3.4.5. Top-k Computation of Minimal Matching Subgraphs

Top-k processing has been proposed to reach early termination after obtaining the top-k results, instead of searching the data graph for all results (see [BHN+02], [HWYY07], [HGP03]). The basic idea originates from the Threshold Algorithm (TA) proposed by Fagin et al. [FLN03]. TA finds the top-k objects with best scores, where the score is computed from the individual scores obtained for each of the object’s attributes. It is required that, for each attribute, the scores are given in a sorted list and the function for the computation of the total score is monotonic. Through (random) access, these attribute scores are retrieved from the list to compute the total object score. Iteratively, the computation is performed for candidate objects that are added to a list. This process continues until the lower bound score of this candidate list (the score of the k-ranked object) is found to be higher than the upper bound score of all remaining objects.

Our method for top-k subgraphs computation is detailed in Alg. 2. Compared with TA, the object is a matching subgraph q, attributes correspond to connections from graph elements n to keyword elements Ki and attribute score is measured in terms of the cost of the path between n and Ki. Instead of a score, the function in Section 5.3.3 is used to compute the cost of a subgraph. Thus, the lower bound score corresponds to the highest cost and vice versa, the upper bound score corresponds to the lowest cost. The highest cost of candidates and the lowest possible cost of the remaining objects are computed as follows:

- **Candidate Subgraphs** As mentioned, top-k is invoked at every step of the exploration. First, the element n that just has been visited during the exploration is examined. In particular, the visited paths stored in n are used to verify whether n is a connecting element. This is the case if all n.Ci are not empty, i.e., for every keyword i, there is at least one path in n.Ci that connects n with an element in Ki. Thus, at least one graph can be obtained by merging the paths that have a different keyword element as origin. However, since every Ci might contain several paths, several combinations of paths are possible. All these combinations are computed and the resulting subgraphs are added to the candidate list kCAN (line 5). Since kCAN is sorted according to the cost of subgraphs, the highest cost of the candidate list is simply the cost of the k-element (line 9).
• **Remaining Subgraphs** During exploration, any element might be found to be a connecting element. Thus, any element could be a candidate from which subgraphs can be generated. In fact, even an element already found to be a connecting element through expansions from some cursors, might still generate further candidate subgraphs, when being explored through expansions from some other cursors. Thus, all elements have to be considered in order to keep track of all the remaining subgraphs. However, we know that in order for some element $n$ to become a connecting element (or generate further subgraphs), it must still be visited by some cursor in $LQ$. Thus, the lowest cost of any potential candidate $n$ must be higher than or equal the cost of the cheapest cursor $c$ from $LQ$.

---

**Algorithm 2**: Top-$k$ Query Computation

1. **Input**: $n, kCAN, LQ, k, kTOP$.
2. **Output**: $kTOP$.
3. **if** $n$ is a connecting element **then**
4.  // process new subgraphs in $n$
5.  $C ← cursorCombinations(n)$;
6.  **foreach** $c ∈ C$ **do**
7.   $kCAN.add(mergeCursorPaths(c))$;
8.  **end**
9. $kCAN ← k$-best($kCAN$);
10. highestCost $← k$-ranked($kCAN$);
11. lowestCost $← \minCostCursor(LQ).w$;
12. **if** highestCost $< lowestCost$ **then**
13.  **foreach** $q ∈ kCAN$ **do**
14.   // add query computed from subgraph
15.   $kTOP.add(mapToQuery(q))$;
16.   **end**
17. // terminates after top-$k$ have been computed
18. return $kTOP$;

Top-$k$ results have been obtained if the highest cost of the candidate list is found to be lower than the lowest cost of the remaining objects. From the candidate list $kCAN$, $k$ top-ranked subgraphs are retrieved. Every subgraph is then mapped to a query (line 14) and results in $kTOP$ are finally returned (line 17).

Compared with related approaches for keyword search, the top-$k$ al-
algorithm discussed above supports general subgraphs and is not limited to trees. Typically, only distance information is incorporated into top-$k$ processing (see [HWYY07], [KPC+05]), while our approach builds on a variety of cost functions. Indexing this information a priori can improve the efficiency of graph exploration as well as top-$k$ processing. This information helps to choose the vertex with the minimal distance to a keyword vertex at every step of the exploration. In fact, it has been shown in [HWYY07] that the results of such a guided-exploration coincide with top-$k$ using the TA algorithm. However, while such an approach guarantees minimality of top-$k$ results w.r.t. a distance metric, it is not straightforward to support scores that cannot be determined a priori. In our approach, minimality can be guaranteed for any score metrics, given that the scoring function is monotonic.

More similar to our approach is the top-$k$ algorithm in [BHN+02], which is also based on TA. The highest cost is also computed using a candidate list. The lowest cost is simply derived from a queue containing the vertices that have not been explored. The authors notice that this is only a coarse approximation since answer trees with higher scores can still be generated with visited vertices. Thus, the author proposes to maintain also the shortest path to a visited element, for all keyword elements. However, due to the distinct root assumption, vertices that are already part of an answer tree are not considered. Note that in our approach, the computation of the lowest cost incorporates any graph elements, i.e., visited elements including the connecting elements. Based on Proposition 1, which basically guarantees that paths are explored in ascending cost order, we can prove that Alg. 2 indeed returns the list of the $k$ minimal matching subgraphs.

**Proposition 2** Let $\mathcal{R}$ be the graph-structured resource model and $K = \{k_1, ..., k_i\}$ be a set of keywords. Then the list of results kTOP produced by the Alg. 1 contains the $k$ queries corresponding to the matching subgraphs of $\mathcal{R}$ with the least costs in ascending order.

**Proof Sketch** Let $c$ be the cheapest cursor remaining in $LQ$ at the time the algorithm terminates and let $k_i$ be the according keyword. Note that the algorithm terminates only if $\text{highestCost} < \text{lowestCost}$, i.e., if the cost of the $k$th-cheapest matching subgraph stored in $kCAN$ is smaller than $c.w$.

Now consider any matching subgraph $q$ of $\mathcal{R}$ with cost cheaper than the cost of the $k$th-cheapest matching subgraph stored in $kCAN$. Let $n$
be the connecting element of this subgraph. Now we know due to the abovementioned termination condition that the cost of $R$ must be smaller than $c.w$, implying that every path from $n$ to the corresponding keyword elements must be cheaper than $c.w$. But then in view of Proposition 1, we can conclude that cursors for all those paths have been created before and hence, $n$ must have been identified as a connecting element. Therefore, all path combinations (particularly the one creating $q$) must already have been added to $kCAN$. So we have shown that any matching subgraph $q$ contained in $R$ having lower cost than the $k$th-cheapest subgraph is contained in $kCAN$ and – since $kCAN$ is sorted – is also contained in $kTOP$ at the time the algorithm terminates.

For the complexity of the exploration, the following worst case complexities can be established: the overall number of cursors to be processed (and hence the time needed) is bounded by $|S_Q|^{d_{\text{max}}}$. This is the maximum number of paths of length $d_{\text{max}}$ that can be discovered during exploration. Moreover, the space complexity is bounded by $k \cdot |K| \cdot |S_Q|$, because for any graph element $n$ from $S_Q$ and any keyword from $K$, at most $k$ cursors have to be maintained, namely those representing the $k$ cheapest paths from $n$ to the respective keyword elements. Note that $|S_Q|$ refers to the size of the query space which tends to be orders of magnitude smaller than the data graph. Yet, there are theoretical cases where the query space is as large as the data graph, hence no better worst case complexities can be guaranteed when referring to the size of the original data graph.

5.3.4.6. Mapping Subgraphs to Queries

In this step, the subgraphs as computed previously are mapped to conjunctive queries. Note that exploration has been performed only on the query space. Thus, edges of subgraphs must be of the form $e(v_1, v_2)$, where $e \in L_R \cup L_A$ and $v_1, v_2 \in V_S \cup \{\text{Thing}\} \cup V_V^R \cup \{\text{Value}\}$. Further, $v_1 \in V_C^S \cup \{\text{Thing}\}$ and $v_2 \in V_V^R \cup \{\text{Value}\}$ if $e \in L_A$, and $v_1, v_2 \in V_C^S \cup \{\text{Thing}\}$ if $e \in L_R$. A complete mapping of such a subgraph to a conjunctive query can be obtained as follows:

- **Processing of Vertices** Labels of vertices might be used as constants. Thus, vertices are associated with their labels such that $\text{constant}(v)$ returns the label of the vertex $v$. Also, vertices might stand for variables. Every vertex is therefore also associated with a distinct variable such that $\text{var}(v)$ returns the variable representing
- **Mapping of A-edges** Edges \( e(v_1, v_2) \) where \( e \in L_A \) and \( v_2 \neq Value \) are mapped to two query predicates of the form \( \text{type}(\text{var}(v_1), \text{constant}(v_1)) \) and \( e(\text{var}(v_1), \text{constant}(v_2)) \). Note that \( e \) is an A-edge, s.t. \( v_1 \) denotes a class and \( \text{constant}(v_1) \) returns a class name. In case \( v_2 = Value \), \( e(v_1, v_2) \) is mapped to the predicates \( \text{type}(\text{var}(v_1), \text{constant}(v_1)) \) and \( e(\text{var}(v_1), \text{var}(Value)) \).

- **Mapping of R-edges** Edges \( e(v_1, v_2) \) where \( e \in L_R \) are mapped to three query predicates of the form \( \text{type}(\text{var}(v_1), \text{constant}(v_1)) \), \( \text{type}(\text{var}(v_2), \text{constant}(v_2)) \) and \( e(\text{var}(v_1), \text{var}(v_2)) \). Note that since \( e \) is an R-edge, \( v_1, v_2 \) denote classes.

By processing the vertices and by the exhaustive application of these mapping rules, a subgraph can be translated to a query. The query is simply a conjunction of all the predicates generated for a given subgraph.

As opposed to related approaches, we compute queries instead of answers. Commonly, answers are assumed to be the roots of some trees found in the data graph (see [BHN+02], [HWYY07], [KPC+05]). This is reasonable if there are no intermediate elements between the root and the keyword elements. Otherwise, also intermediate elements have to be considered as candidate answers since just like the root, they might be relevant for the user’s information need. These candidate answers cannot be retrieved using current techniques. Further, candidate answers might be part of different trees having the same root. These answers are left out due to the distinct root assumption.

We compute all different substructures that can connect keywords. Since queries are derived from these substructures, the underlying query engine can be leveraged to retrieve all answers for a given query. If there is no further information available other than keywords, a reasonable choice is to treat all query variables as distinguished to obtain all variable substitutions of a given query. In the final presentation of the queries, specific mechanisms can be provided for the user to choose the distinguished variables.

### 5.3.5. Comparison to Related Work

Throughout the section, we have discussed the most relevant related work, namely the dataguide concept [GW97a], IR-based scoring metrics (e.g. [GSW05, HGP03, HHP06, LYMC06, GSBS03, AYKM+05]), basic search algorithms for graph exploration ([HWYY07, BHN+02, KPC+05]) and
the Threshold Algorithm [FLN03], the fundamental algorithm for top-$k$ processing. We will now present a broader overview of related work on keyword search, and further information on graph exploration and top-$k$.

5.3.5.1. DB-style Keyword Search

There exists a large body of work on keyword search on structured data (see [ACD02, BHN+02, HP02, HGP03, LYMC06]). Here, native approaches can be distinguished from the ones that extend existing databases with keyword search support. Native approaches support keyword search on general graph-structured data. Since they operate directly on the data, these approaches have the advantage of being schema-agnostic. However, they require specific indices and storage mechanisms ([BHN+02, KPC+05, HWYY07]) for the data. Database extensions require a schema, but can leverage the infrastructure provided by an underlying database. Example systems implemented as database extensions are DBXplorer [ACD02] and Discover [HP02]. These systems translate keywords to candidate networks, which are essentially join expressions constructed using information given in the schema. These candidate networks are used to instantiate a number of fixed SQL queries.

Our approach combines the advantages of these two approaches: in line with native approaches, it is also schema-agnostic. The semantic model is derived from the data to capture the “schema” information that is necessary for query computation. Unlike the natives approaches, the exploration does not operate directly on the data, but on the semantic model. In addition, our approach can leverage the storage and querying capabilities of the underlying database engine. A main difference to both the mentioned types of approaches is that, instead of computing answers, we generate top-$k$ queries. Instead of mapping keywords to data tuples, we map keywords to elements of a query. In this way, more advanced access patterns can be supported. In particular, keywords are treated as terms in previous approaches, while they might be recognized also as query predicates in our approach. This querying capability can be further extended by introducing special elements in the semantic model that represent additional query constructs such as filters. Besides, the presentation of queries to the user can facilitate comprehension and further refinement. All variable bindings are retrieved for a chosen query, instead of the roots of the top-$k$ trees only (like in [HWYY07, BHN+02]).
5.3.5.2. Graph Exploration

There are algorithms for searching substructures in tree-structured data (see [FKM00, CMKS03, GSBS03, LYJ04]). More related to our work are algorithms on graphs, particularly backward search [BHN+02] and bidirectional search as discussed previously. Since the exploration of a large graph data is inherently expensive, dedicated indices are proposed to store not only keyword elements but also specific paths [GSBS03] or connectivity information of the entire graph [HWYY07]. While our approach can benefit from indexing distance information (for a more guided exploration), its application is limited to scores that can be computed off-line. The crucial difference is that while existing algorithms compute trees with distinct roots only, our search algorithm computes general subgraphs. For this, we need to traverse both incoming and outgoing edges as well as keep track of all possible distinct paths that can be used to generate subgraphs.

5.3.5.3. Graph Data Scoring

For ranking answers, many possible strategies have been suggested to take information from both the content (labels) and the structure of the graph into account. In particular, several approaches (see [GSW05, HGP03, HHP06, LYMC06, GSBS03, AYKM+05]) have adopted IR-style ranking to compute scores for answer trees. Sophisticated ranking has not been our primary concern. Compared to these approaches, the metrics used here are rather simple, putting emphasis on the efficiency of computation. Yet, the employed scheme is modular and can flexibly incorporate different metrics. For instance, the notion of PageRank as proposed in [HHP06] can be applied to replace the current metric for popularity.

It is common to distinguish query independent from query specific metrics. Approaches that store connectivity and distance information as discussed above do not support query specific metrics because it is hard to update the index accordingly. This type of metric is however crucial since the query captures the users’ intention. In this approach, this score is computed w.r.t. both syntactic and semantic similarity, using a customized keyword index implemented on top of an IR engine.

5.3.5.4. Top-\(k\) Graph Computation

We have discussed that a perfectly guided exploration using pre-indexed distance information [HWYY07] can lead to results with best scores. This is however only possible with scores that can be derived from pre-indexed
information. Top-$k$ algorithms that compute scores online typically rely on TA (e.g. [BHN‘02, KPC‘05, HGP03]). Compared to our top-$k$ procedure, these algorithms compute tree structures only and most importantly, rely on heuristics such that no top-$k$ guarantee can be provided for the final results.

5.4. An Empirical Study of Query Construction and Refinement

In this section, we report on experiments that have been performed with different schema-agnostic search approaches. We will first discuss the results achieved with SemSearchPro. Then, we present a task-based study conducted to assess the applicability of other approaches.

5.4.1. Evaluation of Query Construction in SemSearchPro

We have implemented the SemSearchPro translation approach and made it available through various systems (see Section 4.4 in Chapter 4). We performed the experiments using Q2Semantic, which is basically an early version of the Hermes system we have presented in Section 4.4. Q2Semantic features a Google-like keyword query interface, which can assist the user in typing keywords. For the entered keywords, it computes the top-$k$ conjunctive queries, transforms them to simple natural language questions, and presents them to the user. In addition, the graph data (connected sub-graphs) that has been explored by the algorithm is visualized to enable query refinement through drag-and-drop.

We now discuss the experiments we have performed to assess the effectiveness and efficiency of SemSearchPro’s query translation. We use DBLP, a dataset containing 26M triples about computer science publications that has been commonly used for keyword search evaluation (see [KPC‘05, HWYY07]). Additionally, TAP and LUBM have been employed to increase the validity of our experimental results. TAP is an ontology of the size of 220k triples, published by the Stanford University. It describes knowledge about sports, geography, music and many other fields. LUBM is the Lehigh University benchmark commonly used in the semantic web community. We use LUBM(50,0) which describes fifty universities. Experiments are conducted on a SMP machine with two 2.0GHz Intel Xeon processors and 4GB memory.

http://tap.stanford.edu  
http://swat.cse.lehigh.edu/projects/lubm/
Table 5.1. Query examples for DBLP.

<table>
<thead>
<tr>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q11 1999 acm</td>
</tr>
<tr>
<td>Q12 computer graphic acm</td>
</tr>
<tr>
<td>Q13 semantic tagging aaai</td>
</tr>
<tr>
<td>Q14 1999 springer</td>
</tr>
<tr>
<td>Q15 database 1999</td>
</tr>
<tr>
<td>Q16 ai 1999</td>
</tr>
<tr>
<td>Q17 ir 1999</td>
</tr>
<tr>
<td>Q18 machine learing 1999</td>
</tr>
<tr>
<td>Q19 1999 isca</td>
</tr>
<tr>
<td>Q20 Web Services 1999</td>
</tr>
<tr>
<td>Q21 Gaussian Weighted Histogram ACM</td>
</tr>
<tr>
<td>Q22 computer graphic IEEE</td>
</tr>
<tr>
<td>Q23 1998 IEEE</td>
</tr>
<tr>
<td>Q24 Gaussian Weighted Histogram IEEE</td>
</tr>
<tr>
<td>Q25 Gaussian Weighted Histogram springer</td>
</tr>
</tbody>
</table>

5.4.1.1. Effectiveness Study

In order to assess the effectiveness of the approach, we have asked a selection of computer scientists to provide keyword queries along with the NL description of the underlying information need. 12 people participated, resulting in 30 different queries for DBLP and 9 for TAP. Table 5.1 lists 15 of the queries used for DBLP. The first 10 queries are also used in [HWYY07]. An example query is “algorithm 1999” and the corresponding description is “All papers about algorithms published in 1999”. We shown TAP queries in Table 5.2, along with the natural language description for each query.

For assessing the effectiveness of the generated queries and their rankings, a standard IR metric called Reciprocal Rank (RR) defined as $RR = 1/r$ is used, where $r$ is the rank of the correct query. According to our problem definition, a query is correct if it matches the information need (the provided NL description). If none of the generated queries match the NL description, RR is 0.

Fig. 5.4.1.1 shows the Mean RR (MRR, the average of the RR scores obtained from the 12 participants) for DBLP, which we have calculated using the scoring functions $C_1$, $C_2$ and $C_3$ as discussed in the previous section.
We observe that some queries such as Q2, Q4, Q6, Q9 and Q10 get rather good results even though only the path length is used for scoring ($C_1$). This is because in these cases, the exploration results in a low number of alternative substructures and queries, respectively. When many substructures can be found, $C_2$ seems to be more effective as it enables the exploration to focus on more “popular” elements. Clearly, MRR obtained using $C_2$ is at least as high as MRR obtained using $C_1$, for all 30 queries. However, MRR is low for $C_2$ when ambiguity introduced through the keyword-to-element matching is high. That is, there are many keywords that match several graph elements, such as in Q4, Q6, Q9, and Q10. Incorporating the matching relevance of keywords helps to prioritize elements that more likely match the user information need. The results show that $C_3$ is superior in all cases.

We get similar conclusions in the evaluation with TAP. The detailed results are shown in Fig. 5.4.1.1.

The overall results indicate that using syntactic similarity and lexical knowledge from WordNet helps to bridge the gap between keywords and labels of data elements. However, this might result in higher ambiguity. Nevertheless, the quality of the produced queries and their rankings is promising.

During this experiment, we have also collected user feedbacks on the usability of our approaches. In particular, we offer the 12 participants two modes for querying: (1) the standard approaches which simply return the top-$k$ results (2) and our approach where the NL questions derived from the generated queries are presented. While many participants think that the generated NL questions are not intuitive, 90 percent prefer to obtain
Table 5.2. Query examples for TAP.

<table>
<thead>
<tr>
<th>Query ID</th>
<th>Keywords</th>
<th>Potential information need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Texas, Nash</td>
<td>Which basketball team located in Texas is Nash playing for?</td>
</tr>
<tr>
<td>Q2</td>
<td>Nash, Louis, Blues</td>
<td>Which Ice Hockey Team located in Louis Blues is Nash playing for?</td>
</tr>
<tr>
<td>Q3</td>
<td>Supergirl</td>
<td>Who is called “supergirl”?</td>
</tr>
<tr>
<td>Q4</td>
<td>Professor, object modeling, universal library</td>
<td>Find professors who wrote a book on object modeling and are involved in a project about universal library.</td>
</tr>
<tr>
<td>Q5</td>
<td>Strip, Las Vegas</td>
<td>What is the well-known “Strip” in Las Vegas?</td>
</tr>
<tr>
<td>Q6</td>
<td>Shanghai</td>
<td>What does “Shanghai” mean?</td>
</tr>
<tr>
<td>Q7</td>
<td>SVG, Capin</td>
<td>Find specification about SVG from an author with last name “Capin”.</td>
</tr>
<tr>
<td>Q8</td>
<td>Nobel, hungry stones</td>
<td>Who won the Nobel Prize and wrote a book named “…hungry stones . . .”?</td>
</tr>
<tr>
<td>Q9</td>
<td>Web Accessibility Initiative, www-rdf-perllib</td>
<td>Find persons who work for the Web Accessibility Initiative and are involved in the activity with mailing list “www-rdf-perllib.”</td>
</tr>
</tbody>
</table>

the questions first, rather than the answers directly. Also, all users prefer to do refinement on the structured query, rather than on the keywords. These initial feedbacks can be seen as an indication for the usability of query presentation. A more sophisticated usability study of this and other schema-agnostic search approaches is presented in the next section. Here, we focus on algorithmic aspects of keyword search and will now continue with the performance analysis of the proposed algorithms.

5.4.1.2. Performance Evaluation

The experiment was performed using the same DBLP data set and the queries that have been used for evaluating the bi-level indexing and query processing scheme [HWYY07]. This approach, called BLINKS, has shown to be faster than the state of the art technique used to deal with keyword queries, namely the bidirectional search approach [KPC+05]. It is based on the idea of indexing all possible paths that may exist between keyword elements. Since the space needed to index the entire data graph might be large, further strategies have been proposed to partition graphs into blocks.

**Query Performance** Strictly speaking, the approaches in this experiment are not directly comparable as previous work directly outputs an-
answers while our translation approach produces queries. Nevertheless, we aim to derive some comparative conclusions using the following setup: we measured the time our approach needed for computing queries as well as the time needed for processing them to obtain the answers. Thus, we consider both query translation and query processing to obtain the same results as the baseline approaches. Precisely, the total time is the time for computing the top-10 queries for the given keywords, plus the time for processing these queries to obtain the top-10 answers.

Fig. 5.11 shows the results for the top-10 answers. Besides our approach, we measured the times needed by the bidirectional approach as well as four configurations of BLINKS: two parameters for block sizes (1000, 300) and two partitioning algorithms (BFS- and METIS-based).

According to these results, our approach outperforms bidirectional search by at least one order of magnitude in most cases. It also performs fairly well when compared with the various configurations of BLINKS. In particular, our approach achieves better performance when the number of keywords is large (Q7-Q10). This suggests that our approach is superior when the queries are more complex, requiring the exploration of complex structures.

It is important to note that these results were achieved without computing and indexing all possible connections between keyword elements. Our approach is merely based on the exploration of a relatively small semantic model, while BLINKS requires a large index to “materialize” and store all possible paths.

**The Effect of Top-\(k\) and Query Length** We have also investigated the impact of top-\(k\) on search performance. Fig. 5.12(a) shows that the
average search time (ms) for 30 queries (keyword query length 2-4) on DBLP varies at different \( k \). It can be observed that the times increase linearly when \( k \) becomes larger. In addition, the impact of query length on the search performance is minimal when \( k \) is 10. The impact of query length is substantial when a large number of results have to be returned (e.g. \( k > 20 \)).

**Index Performance** Since the efficiency largely depends on the size of the semantic model, we also analyzed the employed indexes. Fig. 5.12(b) shows that the size of the keyword index (inverted index) is very large for DBLP. DBLP has much more V-vertices than LUBM and TAP. This indicates that the size of the keyword index is largely determined by the number of V-vertices in the data graph. However, the size of the semantic model index (graph index) rather depends on the structure of the data graph and the number of edge labels and classes. TAP has much more classes than LUBM and DBLP, resulting in a much larger semantic model index.
5.4.2. Evaluation of Schema-agnostic Construction & Refinement Approaches

In this section, we present a comparison of the different search paradigms presented before using an experimental study. Because of the types of complex information needs we are dealing with (relation search in particular), the standard evaluation based on the metrics of precision and recall is too limited. Given complex needs, we aim to assess whether users are able to address them and how much effort they have to invest. Thus, we conduct a task-based evaluation [ER07] which has gained acceptance in the IR community – especially for dealing with search solutions that go beyond the standard document- and keyword-centric IR paradigm. We compare the search paradigms in terms of effectiveness and efficiency, and finally, we discuss some initial usability results.

5.4.2.1. Description of the Experiment

Every participant had to use every one of the search paradigms to solve nine tasks of varying complexity.

Tasks The nine tasks involved the types of search discussed before, i.e., three of them rely on entity search, another three rely on fact search, and the remaining three involve relation search. Table 5.4.2.1 shows the tasks for one of the groups of the user study. Every one of the tasks was in principle solvable with either one of the search paradigms. The process of task execution was monitored by an expert. We considered a task as correctly solved as soon as the answer to the task was displayed on the screen, and the participant was able to identify this answer. For every task, the participant had the possibility to decline it as not solvable with an arguable effort. If a user spent more than three minutes trying to solve a task, we aborted this task and considered it not solved.

Participants We conducted the experiment with 19 volunteers at the age of 21 to 37 years. Nine of them were software developers, two were computer science researchers and the rest of the participants were non-technical users. Fig. 5.13 shows statistics about the self-assessment of the participants. Most of them had experience with query languages like SQL or XQuery. The experience with standard Semantic Web Technologies like RDF or SPARQL was almost equally distributed from very much experience to no experience at all.

Systems We implemented the search paradigms and integrated them as separate search modules into a demonstrator system of the Information
Table 5.3. Tasks of group A.

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Find Tom Hanks (the actor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 2</td>
<td>Find The Beatles (the band)</td>
</tr>
<tr>
<td>Task 3</td>
<td>Find Boston (the place)</td>
</tr>
<tr>
<td>Task 4</td>
<td>Find the occupation of Barack Obama</td>
</tr>
<tr>
<td>Task 5</td>
<td>Find the hometown of Metallica</td>
</tr>
<tr>
<td>Task 6</td>
<td>Find the birthplace of Jesus</td>
</tr>
<tr>
<td>Task 7</td>
<td>Find people with birthplace Karlsruhe</td>
</tr>
<tr>
<td>Task 8</td>
<td>Find the names of the spouses of all directors of Rambo movies</td>
</tr>
<tr>
<td>Task 9</td>
<td>Find all people whose birthplace is Albany, together with their deathplace</td>
</tr>
</tbody>
</table>

Figure 5.13. (a) Skills in using search engines and (b) experience with different technologies.

Workbench (presented in Section 4.4) that has been developed as a showcase for interaction with the Web of data. In particular, keyword search is implemented according to the design and technologies employed by standard Semantic Web Search engines. Like Sindice and FalconS, we use an inverted index to store and retrieve RDF resources based on terms. Also using the inverted index, faceted search is implemented based on the techniques discussed in [BYGH+08]. Result completion is based on recent work discussed for the TASTIER system [LJLF09]. For computing join graphs, we use the top-k procedure elaborated in [TWRC09]. This technique is also used for computing top-k interpretations, i.e., to support query completion. We choose to display the top-6 queries and the top-25 results respectively.

Dataset For the evaluation, we used DBpedia, a dataset covering a large amount of broad-ranging knowledge [BLK+09] together with YAGO [SKW08]. It allowed us to design evaluation tasks that are not domain
Chapter 5. Query Construction and Refinement

5.4.2.2. Evaluation results

In this section we firstly present the effectiveness and efficiency results. Then, we discuss usability issues based on results of the user questionnaire.

**Effectiveness** In our context, we define the effectiveness as the fraction of the tasks which were solved correctly. The percentage of unsolved tasks grouped according to the classification of information need discussed in Section 4.2.2 of Chapter 4 is shown in Fig. 5.14. We will now discuss the results with respect to the types of search used to solve the tasks.

The results show that the paradigms do not differ for simple entity search. Every participant was able to solve every entity search task with every paradigm. This is not very surprising, since entity search can mostly be performed by simply typing the name of the entity. Then, the user conducts keyword search directly or obtains a translated query and evaluates specific.

The experiments were conducted using commodity PCs (i.e., Intel Pentium Core2Duo with 2 x 2Ghz CPU and 3 GB RAM). All actions taken by the participants and the system responses were recorded using a screencast software. The experimental study was based on both the analysis of these screencasts as well as a questionnaire every participant had to answer after the experiment. Additional information such as this questionnaire and the handouts with a list of the tasks provided to participants can be found in [Mat09].

Figure 5.14. Percentage of unsolved tasks.
this query. Either way, it was straightforward for participants to accomplish all tasks.

For fact search, we noticed slight differences between the paradigms. About 2 percent of the fact search tasks were not solved with keyword search and faceted search. This means that one of the participants did not accomplish one task. We observed more failures with result completion: 9 percent of the fact queries were not solved with this paradigm. A problematic task was for instance *Find the birthplace of Jesus*. In this case, the result set is very large such that the item of interest (i.e., the entity Jesus) is not part of the top-25 results. However, more tasks were solved with query completion. More of the item of interests were covered by the top queries than the top results.

The differences in effectiveness between the paradigms became obvious for relation search: while only one candidate (about 5%) had problems obtaining the answers for relation search using query completion, we noted the percentage of tasks not solved using result completion to be 25 percent. The tasks that were difficult are for instance *Find people whose birthplace is Washington, together with their death place*. Since this involves a very special information need it was necessary to specify very precise keywords. Otherwise, the number of results computed as candidate answers is high – and in this case, does not contain the relevant answer. We found that with relation search, the percentage of unsolved talks is even higher for keyword search. For instance, only one candidate was able to accomplish searching for people born in Washington and their death place. To do this, users have to find the town (i.e., Washington) first. This is easily done by submitting the keyword *Washington*. Then, users have to inspect the information contained in the resource page. In particular, they have to find all the people born in this city. This is also relatively straightforward because in this case, this information was simply given in a list. However, much effort is required in the final step, where users have to visit the pages of every person in order to find out their death place. Along the way, most participants recognized this problem and did not complete or refused to complete the task because the given time of three minutes is too limited.

**Efficiency** For assessing the efficiency, we identified all basic user interactions with the systems. These interactions correspond to the steps as discussed previously for the search process. We consider the operations of entering a keyword, executing the keyword query (time needed for the keyword search step in keyword search and faceted search), completing
Chapter 5. Query Construction and Refinement

Figure 5.15. (a) Number of basic operations, (b) average time users needed to solve a task (in seconds).

The efficiency results for the different paradigms are shown in Fig. 5.15. The results suggest the time needed for keyword search and faceted search increases rapidly with the complexity of the information needs. This is clearly illustrated in Fig. 5.15(b). While keyword search and faceted search were the most efficient paradigms for entity queries, the time almost doubled for fact queries. This trend continues as we turn to the even more complex conjunctive queries. We observed another large increase of 40 to 60 percent. The increase in time invested by the user is in fact larger for faceted search than for keyword search. Fig. 5.15(a) delivers an explanation for this: The number of basic operations performed by participants using faceted search is by far the highest. Although each can be performed efficiently, the overall high number of operations lead to a large amount of time in total.

In contrast to these two paradigms, we did not observe a significant increase of time for the completion-based paradigms. In fact, the time invested by participants is almost the same for all types of information needs, as illustrated in Fig. 5.15(b). Fig. 5.15(a) also shows that the number of basic operations is independent of the complexity of the information
needs. The search process consists of the same steps for every query: enter a list of keywords, translate those keywords, and, when using query completion, choose the correct query, and finally, find the answer in the result set. In the cases users did not choose the keywords precisely enough, they had to repeat the entire process.

Under the aspects of efficiency, we observed only minimal differences between result and query completion. Result completion seems to be faster in some cases.

User Questionnaire After the experiment, we conducted an interview with all of the candidates to assess the usability of the search paradigms. Fig. 5.16(a) shows that 12 out of 19 candidates (65%) liked the query completion most. The preferences of the remaining candidates were equally distributed among the other three paradigms. Further details on the comparison of the two paradigms with best evaluation results so far systems are shown in Fig. 5.16(b+c). It seems that most users did not face any difficulties with choosing the correct results or the correct queries while using the completion-based paradigms. Query completion seems to be slightly more usable than result completion.

5.5. Conclusions

The increasing availability of structured and semantic data on the Web bears potential for addressing complex information needs more effectively. A primary challenge lies in enabling the users to specify complex queries without requiring them to know details of the internal data and query model, and the underlying schema of the data. In this section, we identified and analyzed the keyword-driven schema-agnostic search paradigm which addresses this challenge. We have conducted a system-
atic study of four popular schema-agnostic approaches that rely on the use of keyword queries: (1) IR-style keyword search, (2) faceted search and DB-style keyword search in the form of (3) result completion and (4) query completion.

We have presented the SemSearchPro approach as one for DB-style keyword search. It translates keyword queries to structured conjunctive queries. Instead of computing answers, novel algorithms for the top-\(k\) exploration of matching subgraphs have been proposed. They operate on a small summary of the data that draws from an automatically constructed semantic model. In the evaluation, keyword search with SemSearchPro achieved good performance when compared to the state of the art. Also, effectiveness results measured in terms of mean reciprocal rank suggest that all queries can be translated correctly using any of the three proposed ranking schemes. The third scheme which incorporates a combination of factors did best.

Another novel concept advocated by the SemSearchPro approach is the explicit use of structured queries. One additional intermediate step has been introduced to the keyword search process, where instead of results, structured queries are presented to the user first. For instance, computed queries can be used to implement query completion, allowing users to select the appropriate interpretations upon entering keywords. Advantages of this concept are many-fold: (1) a structured query can serve as result description, (2) it can lead directly to the desired results corresponding to the intended interpretation and also, (3) it can be refined more precisely than using a keyword query.

In a broader context, we have studied the quality and usability of schema-agnostic search approaches – including SemSearchPro – for query construction and refinement, from a process-oriented view and then performed a controlled user study to compare them. From our experimental study we can draw the following conclusions: not surprisingly, IR-style keyword search turned out to be sufficient for simple information needs. In fact, the results show that the effectiveness of the approaches does not differ for simple entity search. The advantages of more advanced search approaches were apparent for more complex information needs. For the most complex information needs under consideration (i.e., relation search), it turned out that with query completion, users can answer most queries in reasonable times.
Chapter 6

Query Processing

6.1. Introduction

This chapter deals with the second step of the process, namely query processing. Basically, this task amounts to matching the query against the underlying resource descriptions. It is executed by the engine after the information need has been specified and become available in the form of a structured query. Thus, it is in fact structured query processing. Semantic data on the Web might be fully-structured as well as semi-structured. Recall that semi-structured data lacks schema information. This chapter is concerned with approaches that can deal with this kind of data that is not associated with a schema – hence the name schema-agnostic query processing approaches. This capability is essential to exploit the possibly larger amount of semi-structured data on the Semantic Web.

Fig. 6.1 illustrates the topics and the structure of this chapter. Section 6.2 provides a brief survey of existing schema-agnostic query processing approaches. In particular, techniques for computing structure indexes from semi-structured and XML data will be discussed. Basically, a structure index compactly represents the structure of the underlying data and thus, can act as a pseudo-schema. The indexes included in this survey are the dataguide proposed for semi-structured data, and the 1-index as well as the A(k)-index proposed for XML data. These indexes enable schema-agnostic query processing in the sense that queries can be processed using the pseudo-schema. Further, the use of structure indexes for improving the performance of query processing has been studied for the tasks of path- and twig-pattern matching. Existing approaches that employ structure indexes for this kind of optimization are also covered in the survey of Section 6.2.
The SemSearchPro approach to query processing is also built upon the structure index concept. As opposed to existing work, the index employed by SemSearchPro can be constructed for general graph-structured data. Thus, it can be used for dealing with the various kinds of data available on the Semantic Web. Section 6.3 discusses the SemSearchPro approach in detail. In particular, it elaborates on how this structure index is constructed and how SemSearchPro uses it as a guide for a structure-based partitioning of the data as well as for structure-aware query processing. Besides this basic approach, an optimization that combines the structure-aware approach (structure-level processing) with standard query processing techniques (data-level processing) is presented in Section 6.3. Experimental results on the index size and the query performance that can be achieved with SemSearchPro’s basic approach as well as its optimization are presented in Section 6.4. This chapter concludes in Section 6.5.

6.2. Schema-agnostic Query Processing Approaches

By schema-agnostic query processing, we refer to the kind of techniques which can cope with semi-structured schema-less data. This is important because an abundant amount of RDF data on the Web, especially the
RDFa data associated with Web pages, are not accompanied by a schema.

In the database community, there is a long tradition for semi-structured data. Different kinds of approaches have been elaborated for indexing, for storing and for processing queries against data without schema information. Basically, for supporting queries on this kind of data, the first thing to do is to construct a pseudo-schema. Structure indexes have been dealt with extensively in the problem domain of semi-structured and XML data management. A structure index describes the structure that is exhibited by the data. Thus, it is conceptually very similar to a data schema. While it has been used for other purposes, this kind of indexes can act as a pseudo-schema, effectively allowing for query processing and for the users to browse the data along the structural dimensions. Thus, enabling schema-agnostic query processing is the first purpose of building a structure index. The second main purpose is the one of optimization. Whether the data is accompanied by a schema or not, a structure index can be built to capture certain structural properties of the data. This information can be leveraged for the task of query processing.

This section aims to provide a brief introduction to the different kinds of structure indexes that have been studied in existing literature. Then, it discusses techniques which exploit them for query processing.

6.2.1. Schema-agnostic Query Processing Using Structure Index

The goal of building a structure index is to capture structural information that can be found in the data. This has been done for both semi-structured data and for fully-structured XML data [BDFS97, KBNK02, KSBG02, QLO03]. Dataguide [GW97b] is one well-known concept that has been proposed for rooted graphs. These are graphs having one distinguished root node from which all other vertices can be reached through directed paths. In particular, the concept of a strong dataguide has been introduced: it is established by grouping together vertices sharing edge label sequences of incoming paths starting from the root vertex. The groups computed this way represent vertices of the data graph. Edges are created between two vertices of the dataguide if there are corresponding edges in the data graph. That is, when there are two connected nodes in the data graph that belong to the two groups represented by these vertices. In fact, this technique is similar to the conversion of a nondeterministic finite automaton into an equivalent deterministic automaton. The grouping in dataguides is not a partition. In other words, one vertex may be assigned
to several groups. The advantage of this determinization is that for any labeled path starting from the root, there is exactly one associated node in the dataguide. However, the size of the dataguide can get exponentially larger than that of the original data graph.

The 1-Index \cite{BDFS97} prevents this worst-case exponential blow-up of a dataguide. The authors suggested to consider the concept of bisimulation. Two nodes are bisimilar when they exhibit the same structure. Generally speaking, these structures constitute sets of incoming and outgoing paths. The 1-indexes can be obtained by grouping bisimilar nodes. The strategy of constructing an 1-index is similar to minimizing a non-deterministic finite automaton. The result is similar to a dataguide with the main difference that it constitutes a partition of the original data graph. Specifically, every node in the data graph “belongs to” exactly one node in the index.

To further reduce the index size, the $A(k)$-Index \cite{KSBG02} has been proposed. The authors of this work proposed to relax the equivalence condition (i.e., the bisimilarity) to consider only incoming paths whose lengths are no longer than $k$. Further, the $D(k)$-Index allows for adjusting $k$ to the query workload \cite{QL003}. Instead of considering incoming paths only, also outgoing paths have been investigated for the construction of covering indexes in the work by Kaushik et al. \cite{KBNK02}. For incoming paths, a backward bisimulation is run on the data graph whereas forward bisimulation is used for outgoing paths.

6.2.2. Optimized Query Processing Using Structure Index

As mentioned, a structure index helps in situation where no schema exists. Further, it has been used extensively for optimized query processing. There are two prominent lines of work. One is about using structure index for evaluating path queries. The other involves the more complex task of XML query processing.

Path queries are expression of the form $P_1 \ x_1 \ P_2 \ x_2 \ ... \ P_i \ x_i$ where $x_i$‘s are distinct variable names, and $P_i$‘s are path expressions. Structure indexes such as the dataguide have been used to evaluate this kind of queries more efficiently. The procedure proposed in \cite{MS99} for instance, firstly computes a set of index nodes that match the query. Then, the union of all data elements that belong to these index matches are returned as answers. In other words, the query is evaluated mainly against the structure index. The data graph is only employed to retrieve the final answers.
In XML query processing, there are patterns of selection predicates on elements related by a tree-structure. There are techniques for finding occurrences of one particular tree-structure in XML data called *twig patterns* [ZND+01, BKS02, CLL05], which also leverage the structure index. Typically, the twig pattern is decomposed into binary relationships. These relationships are either of the type parent-child or ancestor-descendant. They are matched against XML data and the results are then combined using structural join such as *multi-predicate merge join* (MPMJ) [ZND+01] and *TwigJoin* [BKS02]. The structural join method uses information encoded in the structure index.

More precisely, matching is achieved in two main steps: (1) match binary relationships against XML data and combine the results using structural join algorithms to obtain basic matches and (2) combine basic matches to obtain final answer. For this first step, a variation of the traditional merge join algorithm has been proposed to deal with multiple predicates. The MPMJ has been proposed for this, which considers both string values and the positional representation of XML elements derived from the structure index [ZND+01]. The main problem with MPMJ is that it may generate unnecessary intermediate results. In worst case, a large number of join results of individual binary relationships does not appear in the final results. Thus, this procedure involves some computation that might be unnecessary. Bruno et al. proposes a holistic twig pattern matching method called TwigJoin that can avoid this [BKS02]. It outputs a list of element paths, where each matches one root-to-leaf path of the twig pattern. When there are only ancestor-descendant edges, this algorithm is optimal such that each of the matches is part of the final answer to the entire twig pattern. In other words, TwigJoin avoids intermediate results unless they contribute to the final results.

The aforementioned approaches assume that every query node is associated with a group of elements of tag $t$. In [CLL05], this is referred to as Tag Streaming. More sophisticated schemes taking the structure of XML elements into account have been studied in [CLL05]. For instance, the *Prefix Path Streaming* (PPS) has been proposed to group elements with the same prefix-path. Using this scheme, also twig-pattern containing parent-child edges can be processed optimally. More precisely, the query processing procedure satisfies the following conditions: (Opt1) Each stream whose elements tag appears in the twig pattern is scanned only once. (Opt2) None of the intermediate paths output during step 1 is redundant. (Opt3) The space required by the algorithm is bounded by a
factor which is independent of the source document size.

6.3. Schema-agnostic Query Processing in SemSearchPro

We will begin with introducing the problem of query processing. Then, we briefly discuss the state of the art of RDF data management, which is a summarization of selected aspects covered by Section 3.4 in Chapter 3. It is presented here for the reader’s convenience and for motivating our work.

6.3.1. The Query Processing Problem

This section deals with the problem of processing queries against the SemSearchPro’s resource model. Recall the this model is basically a data graph, as illustrated in Fig.6.2(a). Recall that the class of queries that are supported by SemSearchPro is of the type conjunctive queries. Fig. 6.2(b) shows an example query. Also, it illustrates that since variables can interact in an arbitrary way, a conjunctive query $q$ is graph structured: it represents a graph pattern $q = (V_{var} \cup V_{con}, L^Q, E^Q)$ consisting of a set of triple patterns $l(v_1, v_2)$ where $v_1$ and $v_2$ might be variables or constant, i.e., $v_1, v_2 \in V_{var} \cup V_{con}$. A solution to $q$ on a data graph representing the resource model $R$ is a mapping $\mu$ from the variables in the query to vertices $e$ such that the substitution of variables in the graph pattern would yield a subgraph of $R$. The substitutions of distinguished variables constitute the answers, which are formally defined as follows:

**Definition 17** Given a data graph $R = (V^R, L, E^R)$ and a conjunctive query $q = (V_{var_d} \cup V_{var_u} \cup V_{con}, L^Q, E^Q)$, where $V_{var_d}$ denotes the set of distinguished variables, $V_{var_u}$ denotes undistinguished variables and $V_{con}$ stands for constant occurring in $q$. Then a mapping $\mu : V_{var_d} \rightarrow V^R$ from the query’s distinguished variables to the vertices of $R$ will be called an answer to $q$, if there is a mapping $\nu : V_{var_u} \rightarrow V^R$ from $q$’s undistinguished variables to the vertices of $R$ such that the function

$$
\mu' : V_{var_d} \cup V_{var_u} \cup V_{con} \rightarrow V^R \begin{cases} 
\mu(v) & \text{if } v \in V_{var_d} \\
\nu(v) & \text{if } v \in V_{var_u} \\
v & \text{if } v \in V_{con}
\end{cases}
$$

satisfies $l(\mu'(v_1), \mu'(v_2)) \in E^R$ for any $l(v_1, v_2)$ in $q$. 
As indicated before, every conjunctive query can be conceived as a kind of graph with variables and constants as vertices and the query atoms as edges. Then it is easy to see that $\mu'$ can be interpreted as a certain type of homomorphism (i.e., a structure preserving mapping) from the query graph to the data graph\(^1\). We will use this perspective of considering matches as homomorphisms in the following sections.

### 6.3.2. The State of the Art in RDF Data Management

Significant efforts have been dedicated to the development of solutions for managing RDF data. Various kinds of triple stores have been developed for storing and querying RDF data such as Hexastore, RDF-3X and SW-Store. They can be distinguished along the following dimensions:

**Data Partitioning** Different schemes have been proposed to govern the ways data is physically organized and stored. A basic scheme is the *triple-based organization*, where one big three-columns table is used to store all triples. Recently, *vertical partitioning* has been proposed to decompose the data graph into $n$ two-columns tables, where $n$ is number of properties [AMMH07].

**Indexing** A popular technique is *multiple indexing*, i.e., creating mult-

\(^1\)As usual, a homomorphism from $\mathcal{R} = (V^R, L, E^R)$ to $\mathcal{R}' = (V'^R, L, E'^R)$ is a mapping $h : V^R \to V'^R$ such that for every $\mathcal{R}$-edge $l(v_1, v_2) \in E^R$ we have an according $\mathcal{R}'$-edge: $l(h(v_1), h(v_2)) \in E'^R$. 

---

**Figure 6.2.** (a) A data graph, (b) a snippet of its semantic model graph (containing only edges with $l \in L_R$) and (c) a query graph for our example, where $u$ stands for undistinguished and $d$ stands for distinguished variable.
tiple indexes for the data for supporting different lookup patterns. The scheme with the widest coverage of access patterns is used in YARS [HD05]. In [WKB08], sextuple indexing has been suggested, which generalizes the strategy in [HD05] such that for different access patterns, retrieved data comes in a sorted fashion.

**Query Processing** Matching a query against the data graph is typically performed by retrieving triples from the data graph and joining them along the query edges. Join processing can be greatly accelerated, when the retrieved triples are already sorted. In fact, the main advantages from vertical partitioning come from the ability to perform fast merge joins on sorted inputs [AMMH07]. Sextuple indexing takes this further to allow this join processing to be applied on different access patterns, e.g. queries with unbound predicates such that $p$ is a variable [WKB08]. Further efficiency gains can be achieved by finding an optimal query plan [NW08].

### 6.3.3. Overview and Main Contributions

These techniques largely improve RDF query processing. They however, rely on schema information – such as for data partitioning [WSKR03, AMMH07]. A schema however, cannot always be found for Web data. Also, the scalable processing of queries at Web scale is still a challenge, especially for complex graph-structured queries. We address these limitations by proposing a new approach for data partitioning and join processing, which leverages a semantic model automatically built from the data. This semantic model corresponds to the bismulation-based model presented in Section 4.3.2 of Chapter 4. Its construction resembles work on structure indexes used in the database community. Thus, in the following, we use the term structured index whenever we refer to the bisimulation-based semantic model. The structure index act as a schema, enabling the effective browsing and querying of schema-less Web data. The contributions and main advantages of our approach are summarized as follows:

**Depth- and Label-Parameterized Structure Index** We propose a structure index which can be computed for general (RDF) graphs to capture different structure patterns exhibited by the data. A structure index is a basically graph, where vertices represent groups of data elements that are similar in structure. It is a generalized concept of the dataguide [GW97b] and the structure indexes used for XML data [BDFS97, KSBG02]. We elaborate on an index construction procedure, which is able to consider structural patterns that have certain labels ($L_1, L_2$) and (or) a certain length
By varying these parameters, indexes of different properties and sizes can be obtained. For instance using \( n = 1 \), vertices of the resulting index are like schema classes, which group together elements that are similar w.r.t. all (incoming and outgoing) edged-labeled paths of length 1. This result is similar to a schema, where a class essentially, is a group of elements which share the same set of edges.

**Workload-optimized Data Partitioning** The properties specified in the schema are used to perform vertical partitioning. We generalize this idea to use the structure index as the basis for a structure-based partitioning approach, i.e., vertices are mapped to physical tables. This is to obtain a contiguous storage of data elements that are structurally similar. With \( n = 1 \), this approach is similar to vertical partitioning where elements with the same property are physically grouped. Using \( n > 1 \) results in more fine-granular groupings of elements. Elements in every group share the same structure. Compared to a vertical table, such a more fine-granular group that can be located for a given query structure, contains more candidate answers. According to the query structures given in the workload, \( n \) as well as \( L_1, L_2 \) can be used to adjust the granularity of the physical groups.

**Integrated Data- and Structure-Level Processing** While the standard approach like [AMMH07] relies only on data-level operations, we suggest to leverage the structure index for query processing. A simple strategy would be to match the query against the structure index first to identify groups of data that satisfy the overall query structure. This is to filter candidates though structure-level processing. Then, these data groups are retrieved and joined. This needs to be performed only for some parts of the query. Thus, this procedure amounts to query pruning and processing the pruned query using standard data-level operations [AMMH07]. Instead of performing structure- and data-level operations successively, and independent from each other, we propose an integrated evaluation strategy that aims at an optimal combination of these operations. The result might comprise structure-level operations only, data-level operations only or a “mix” of structure- and data-level operations.

The remainder of this chapter is structured as follows: Section 6.3.4 introduces the structure-based approach, describing the notion of structure index, the idea behind structure-based partitioning and structure-aware query processing. The optimization of this approach comprising of index parameterization and integrated query evaluation is elaborated in Section 6.3.5. Complexity issues and optimal strategies for parameterization de-
rived from them are presented in Section 6.3.6. We review related work in Section 6.3.7 before presenting experiments along with performance results in Section 6.4 before and conclude in Section 6.5.

6.3.4. Structure-based Approach

The structure index is basically a compact representation of the data graph which can be used for organizing, browsing and querying schema-less data. In this section, we introduce this concept and show how it can effectively guides the partitioning of data and the processing of queries.

6.3.4.1. Structure Index for Graph Structured Data

A structure index for graph structured data such as RDF is a special graph derived from structural patterns found in the data. Vertices of such an index graph essentially stand for groups of data graph elements that are similar in structure, where structure refers to the set of incoming and outgoing connections. More precisely, we define similarity in terms of structural “neighborhood”, using the well-known notion of bisimulation originating from state-based dynamic systems. We consider graph vertices \( v_1, v_2 \) as bisimilar (written: \( v_1 \sim v_2 \)), if they cannot be distinguished by looking at their “neighborhood”. At first, we consider the case of complete bisimilarity, where the neighborhood to be considered is simply the entire graph.

**Definition 18** Given a data graph \( \mathcal{R} = (V^R, L, E^R) \), a (back-and-forth) bisimulation on \( \mathcal{R} \) is a binary relation \( B \subseteq V^R \times V^R \) on the vertices of \( \mathcal{R} \) such that for \( v, w \in V^R \) and \( l \in L \):

- \( vBw \) and \( l(v, v') \in E^R \) implies that there is a \( w' \in V^R \) with \( l(w, w') \in E^R \) and \( v'Bw' \),
- \( vBw \) and \( l(w, w') \in E^R \) implies that there is a \( v' \in V^R \) with \( l(v, v') \in E^R \) and \( v'Bw' \),
- \( vBw \) and \( l(v', v) \in E^R \) implies that there is a \( w' \in V^R \) with \( l(w', w) \in E^R \) and \( v'Bw' \),
- \( vBw \) and \( l(w', w) \in E^R \) implies that there is a \( v' \in V^R \) with \( l(v', v) \in E^R \) and \( v'Bw' \).

Two vertices \( v, w \) will be called bisimilar (written \( v \sim w \)), if there exists a bisimulation \( B \) with \( vBw \).
Based on this notion of bisimilarity, we will define a special notion of extension, which contains all pairwise bisimilar elements. These extensions can be computed for a given data graph using a bisimulation procedure that will be discussed in the next section. They form a partition of the vertices $V^R$, i.e., a family of pairwise disjoint sets whose union is $V^R$. The structure index graph $R_\sim$ of $R$ is defined in terms of extensions and relations between them.\footnote{Note that the definition is given for an arbitrary equivalence relation $\sim$ such that it carries over to adapted notions of bisimilarity introduced later on.}

**Definition 19** Let $R = (V^R, L, E^R)$ be a data graph and $\sim$ some equivalence relation on $V^R$. Vertices of the associated semantic model also called the index graph $R_\sim = (V^R_\sim, L, E^R_\sim)$ are exactly $R$’s $\sim$-equivalence classes $V^R_\sim = \{ [v] | v \in V^R \}$, with $[v] = \{ w \in V^R | v \sim w \}$. Labels of $R_\sim$ are exactly the labels of $R$. An edge with a certain label $l$ is established between two equivalence classes $[v]$ and $[w]$ exactly if there are two vertices $v^* \in [v]$ and $w^* \in [w]$ in $R_\sim$ such that there is an edge $l(v^*, w^*)$ in the data graph $R$, i.e., $E^R_\sim := \{ l([v^*], [w^*]) | l(v^*, w^*) \in E^R \}$.

**Example 17** A structure index for our example data graph is shown in Fig. 6.2(b). For the sake of presentation, the index shown is only a snippet of the complete index, illustrating only structures formed by relation edges, i.e., edges with $l \in L_R$. It shows eight extensions of structurally similar elements that can be derived from these structures. For instance, $p1$ and $p3$ are grouped into extension $E2$ because they share the incoming connections $\text{supervise}$ and $\text{knows}$ and the outgoing connections $\text{knows}$, $\{\text{worksAt, partOf}\}$ and $\{\text{authorOf, conference}\}$.

### 6.3.4.2. Structured-based Partitioning

The idea of structure-based partitioning (SP) is to apply the grouping of elements as captured by vertices of the structure index to the physical organization of data. In particular, we create a physical group for every vertex of the index graph, one for every extension. Basically, every group contains triples that talk about elements of a particular extension. More precisely, triples $\langle s, p, o \rangle$ are in a group $g_{E_i}$ when their subjects $s$ belong to the same extension $E_i$. Recall that extensions represent partitions of the data graph. In other words, a data element is associated with exactly one
extension. Thus, grouping the triples based on extensions this way guarantees an exhaustive and redundancy-free decomposition of data graph elements.

**Example 18** Some groups resulting from partitioning the data graph in Fig. 6.2(a) are shown as extensions in Fig. 6.2(b). The physical group of triples corresponding to $E_2$ is shown in Tab. 6.1.

<table>
<thead>
<tr>
<th>property</th>
<th>subj</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>authorOf</td>
<td>p1</td>
<td>a1</td>
</tr>
<tr>
<td>authorOf</td>
<td>p3</td>
<td>a2</td>
</tr>
<tr>
<td>worksAt</td>
<td>p1</td>
<td>i1</td>
</tr>
<tr>
<td>worksAt</td>
<td>p3</td>
<td>i2</td>
</tr>
<tr>
<td>knows</td>
<td>p1</td>
<td>p3</td>
</tr>
<tr>
<td>knows</td>
<td>p3</td>
<td>p1</td>
</tr>
</tbody>
</table>

**Table 6.1.** Triples for extension E2.

Compared to vertical partitioning (VP), where triples with the same properties are grouped together, the partitioning proposed here results in the contiguous storage of triples that are same in structure. Clearly, whether a triple satisfies not only a single triple pattern but the entire query depends on its structure, i.e., the incoming and outgoing edges resp. paths. Therefore, if they are physically grouped together based on structural similarity, a group identified for a given query structure would contain more candidates. As opposed to vertical partitioning, triples fetched from disk for such a group satisfy not only the predicate of a given triple pattern but also the entire query structure. Thus, fewer candidates have to be retrieved, resulting in reduced I/O costs.

**Example 19** We show example indexes in Fig. 6.3 to illustrate the differences in physical realization. Indexes are created for every group to access its elements. For vertical partitioning, a Property-Subject-Object (PSO) index may be created for the group of triples $\langle s_i, \text{authorOf}, o_j \rangle$, i.e., `authorOf` points to a sorted list of subjects $s_i$, where each entry has a reference to a list of object(s) $o_j$. For structure-based partitioning, indexes are created for extensions. For instance, an Extension-Property-Subject-Object (EPSO) index may be created for $E_4$ to store all triples of the form $\langle s_i \in E_4, p_k, o_j \rangle$. For $p_k = \text{authorOf}$, only triples with $s_i = p1, p3 \in E_4$ are considered for indexing. Note that entries in the
EPSO index are also sorted. Thus, it enables linear merge join just like PSO used for vertical partitioning.

![Example indexes for VP and SP.](image)

**Figure 6.3.** Example indexes for VP and SP.

### 6.3.4.3. Structure-aware Query Processing

For this graph matching problem, the query is processed not directly against the data but against the structure index first. The idea here is to use the index to locate candidate groups of data that satisfy the entire query structure, thereby filtering irrelevant data. Only in the second step, answers matching not only the query structure, but also “explicitly mentioned” query elements, i.e. constants and distinguished variables, are computed from the candidate groups.

The procedure for query processing can be decomposed into 3 main steps: (1) index graph matching, (2) query pruning and (3) final answer computation. At first, the query graph $q$ is matched against the index graph $\mathcal{R}_\sim$ to find matching extensions $E_i$ on $\mathcal{R}_\sim$. This graph matching is performed using the standard processing technique [AMMH07]. This is done by retrieving triples (edges of $\mathcal{R}_\sim$) for every triple pattern of $q$ and then join them along edges of $q$. The difference here is instead of the data graph, the smaller index graph $\mathcal{R}_\sim$ is used. Final results will be derived from the candidate groups $E_i$ resulting from this step. In particular, triples matching the concrete constants and distinguished variables are retrieved from $E_i$, and joined along the edges of $q$. However, this might be performed for only some parts of the query $q$, as other parts might have been discarded during a preceding pruning step.

The proof of soundness and completeness for this procedure is based on the following property that can be derived for the structure index.
**Proposition 3** Let \( \mathcal{R} \) be a data graph with the associated index graph \( \mathcal{R}_\sim \) for some equivalence relation \( \sim \) and let \( q \) be another graph such that there is a homomorphism \( h \) from \( q \) into \( \mathcal{R} \). Then \( h_\sim \) with \( h_\sim(v) = [h(v)] \) is a homomorphism from \( q \) into \( \mathcal{R}_\sim \).

**Proof** Given a \( q \)-edge \( l(v_1, v_2) \), first observe that \( l(h(v_1), h(v_2)) \) is a \( \mathcal{R} \)-edge since \( h \) is a homomorphism. Then we find that \( l(h_\sim(v_1), h_\sim(v_2)) = l([h(v_1)], [h(v_2)]) \) is a \( \mathcal{R}_\sim \)-edge due to the definition of the index graph. \( \square \)

Intuitively speaking, there is a match of \( q \) on \( \mathcal{R} \) only when there is also a match of \( q \) on \( \mathcal{R}_\sim \). Further, the resulting index graph matches \( E_i = [h(v)] \) from \( \mathcal{R}_\sim \) will contain the query answers \( h(v) \). For retrieving the final answers, one another property can be exploited. It is based on the observation that for tree-shaped parts of the query containing only undistinguished variables, no further processing at the data-level is needed. These parts represent structural constraints only, which have already been verified during the previous index matching step.

The following proposition captures this, specifying that given such a query part \( q \), data contained in the index graph extension \( E_r = [h(r)] \) matching the query root node \( r \) represents the final answers. Thus, such query parts can be removed before final answer computation.

**Proposition 4** Let \( \mathcal{R} \) be a data graph with bisimulation \( \sim \) and the associated index graph \( \mathcal{R}_\sim \). Let \( q \) be a tree-shaped query part containing undistinguished variables only with root \( r \). Let \( h_\sim \) be a homomorphism from \( q \) to \( \mathcal{R}_\sim \) with \( h_\sim(r) = [h(r)] \). Then for every data graph node \( v \in h_\sim(r) \), there is a homomorphism \( h \) from \( q \) to \( \mathcal{R} \) with \( h(r) = v \).

**Proof** We do an induction on the maximal tree-depth of \( q \). As base case, note that for tree depth 0, the claim is trivially satisfied. For any greater tree depth, subsequently consider every child node \( v' \) of \( r \) in \( q \). Assume \( l \in L \) and \( l(h_\sim(r), h_\sim(v')) \in E^R_\sim \). Then, by definition of bisimilarity, there must exist a \( w \in h_\sim(v') \) with \( l(v, w) \in E^R \). We chose \( h(v') = w \). Now we invoke the induction hypothesis for the subtree of \( \mathcal{R}_\sim \) with root \( v' \) which yields us the values \( h(u') \) for all successors of \( v' \). The backward case follows by symmetry. So we have constructed \( h \) with the desired properties. \( \square \)

**Example 20** Fig. 6.2(c) depicts a query, which asks for authors \( x \) of things \( y \) at conferences \( v \), who are supervised by \( w \), are of age 29, and work at a place \( z \) called AIFB that is part of KIT. As illustrated in
This query produces one match on the index graph in Fig. 6.2(b), i.e., $h_{\sim_1} = \{z \mapsto E3, u \mapsto E5, x \mapsto E2, y \mapsto E4, v \mapsto E6, w \mapsto E1\}$. Based on this, we know that data elements belonging to extensions obtained from the index graph match satisfy the query structure, e.g., elements in $E2$ are authors of $y$, supervised by $w$, work at some place $z$ etc. Tree-like parts that can be pruned are $\text{supervises}(w, x)$ and $\{\text{authorOf}(x, y), \text{conference}(y, v)\}$. E.g. Since elements in $E2$ are already known to be supervised by some $w$, there is no need to retrieve data from $E1$. Data processing is however needed for the remaining query parts. Fig. 6.5 shows the pruned query and the data involved in this step. Now, we have to look at the data to find out which elements in $E2$ are of age 29, which elements in $E3$ have the name AIFB, which elements in $E5$ have the name KIT, and whether the elements matching these constants are connected over the relations specified in the query. For this, we need to retrieve and join the triples for $\langle x \text{ age 29} \rangle$, $\langle x \text{ worksAt } z \rangle$, $\langle z \text{ name AIFB} \rangle$, $\langle z \text{ partOf } u \rangle$, $\langle u \text{ name KIT} \rangle$. Note that in the extreme cases where no index graph matches can be found, we can skip the entire second step to avoid data access and joins completely.
6.3.5. Optimized Query Processing

The structure-aware processing discussed previously captures the main idea of using the structure index. However, this naive approach has certain drawbacks, which will be discussed in this section. The focus lies on the optimization techniques addressing these drawbacks.

6.3.5.1. Parameterizing the Structure Index

Recall that the structure index introduced previously based on complete bisimilarity. The neighborhood considered is thus the complete graph. This results in fine-granular groupings of elements that exhibit high structural similarity. This means that the number of extensions might be high, resulting in an index graph that is too large in size. Clearly, the performance of structure-level operations is influenced by the index graph size.

**Depth- and Label-Parameterized Structure Index** Based on the idea that instead of the entire graph, only certain information might be considered for bisimilarity, we propose two concepts for index parameterization. One way is to take only certain labels into account. Another way is to restrict the neighborhood size. These concepts are formalized by the notion of \((L_1-L_2)\) label-parameterized bisimulation of bounded depth \((n)\).

**Definition 20** Given a data graph \(\mathcal{R} = (V^R, L, E^R)\), two edge label sets \(L_1, L_2 \subseteq L\) and a non-negative integer \(n\), two vertices \(v, w \in V^R\) will be called \(L_1\)-forward-\(L_2\)-backward \(n\)-bisimilar (written \(v \sim^R_n w\)), iff a) \(n = 0\) or b) \(n > 0\) and

- there is a \(w' \in V^R\) with \(l(w, w') \in E^R\) with \(l \in L_1\),
- there is a \(v' \in V^R\) with \(l(v, v') \in E^R\) and \(v'^{n-1} \sim w'\) for every \(l(w, w') \in E^R\) with \(l \in L_1\),
- there is a \(w' \in V^R\) with \(l(w', w) \in E^R\) and \(v'^{n-1} \sim w'\) for every \(l(v', v) \in E^R\) with \(l \in L_2\) and
- there is a \(v' \in V^R\) with \(l(v', v) \in E^R\) and \(v'^{n-1} \sim w'\) for every \(l(w', w) \in E^R\) with \(l \in L_2\).

Intuitively speaking, this inductive definition characterizes elements \(w\) and \(v\) \(n\)-bisimilar, if \(\mathcal{R}\) contains for every \(n\)-depth tree structure with root \(w\) an equivalent tree-structure with root \(v\) and vice versa. Consequently, from \(w \sim^R v\) follows \(w \sim^R_k v\) for all \(k \leq n\).
Thus, only trees of maximum depth \( n \) are taken into account. Further, \( L_1 \)-forward-\( L_2 \)-backward parameterization specifies that the relevant edge trees for testing bisimilarity are those where all root-leaf paths traverse forward only \( L_1 \)-edges and backward only \( L_2 \)-edges.

It is easy to see that for a given data graph, the number of vertices and edges of the index graph increases with \( n \), i.e., for \( k < n \) we have

\[
1 \leq |V^{R}_{k}| \leq |V^{R}_{n}| \leq |V| \quad \text{and} \quad 1 \leq |E^{R}_{k}| \leq |E^{R}_{n}| \leq |E|.
\]

The same result applies for label parameterization, i.e., index size increases with the number of labels contained in \( L_1, L_2 \). Thus, the size of the index graph can be adjusted via the parameters \( n, L_1, L_2 \).

Besides size, the parameterization has also consequences for query processing. Data contained in index graph matches satisfies only query structures that are of length \( n \) and covered by the labels in the sets \( L_1, L_2 \). Thus, query pruning cannot be applied without restriction. The notion of prunable tree-shaped query parts has to be adopted:

**Definition 21** Every edgeless single-vertex query graph \( \{v\}, L, \emptyset \) is \( L_1 \)-forward-\( L_2 \)-backward tree-shaped with root \( v \). For any two tree-shaped graphs \( G_1 = (V_1, L, E_1) \) and \( G_2 = (V_2, L, E_2) \) with disjoint vertex sets, \( v \in V_1 \) and \( r \in V_2 \) being \( G_2 \)'s root, the graph \( G = (V_1 \sqcup V_2, L, E_1 \sqcup E_2 \sqcup \{e\}) \) where either \( e = l(v, r) \) with \( l \in L_1 \) or \( e = l(r, v) \) with \( l \in L_2 \) is \( L_1 \)-forward-\( L_2 \)-backward tree-shaped.

This inductive definition essentially specifies that a query (part) is \( L_1 \)-forward-\( L_2 \)-backward tree-shaped if its edges interpreted as undirected edges form an undirected tree and from a distinguished variable root node \( r \), every path from this root to the leaves traverses forward only \( L_1 \)-labeled edges and backward only \( L_2 \)-labeled edges. Note that depth-parameterization is implied by this definition. More explicitly, maximum length of such paths can be defined as \( n \).

We will now revisit the properties for the structure index to take parameterizations into account:

**Proposition 5** Let \( \mathcal{R}(V^R, L, E^R) \) be a data graph, with \( n \)-bounded \( L_1 \)-forward-\( L_2 \)-backward bisimilarity \( \sim \) and index graph \( \mathcal{R}_n \) and let \( q \) be a conjunctive query. Then the following hold:

1. if \( \mu \) is a match of \( q \) into \( \mathcal{R} \) then the function \( \pi \) defined by \( \pi(x) = [\mu(x)]_n \) is a match of \( q \) into \( \mathcal{R}_n \).
2. Let \( q \) be \( L_1 \)-forward-\( L_2 \)-backward tree-shaped with depth \( \leq n \) and root \( r \) and let all non-root vertices of \( q \) be non-distinguished vari-
ables. Then, for every match $\pi$ from $q$ into $R_{\sim}$ and every $v \in \pi(r)$ there is a match $\mu$ from $q$ into $R$ with $\mu(r) = v$.

**Proof**

1. From the definition of $\sim$ directly follows that it is an equivalence relation. Hence Proposition 3 applies, which ensures the desired property.

2. We prove this by an induction on the maximal tree depth of $q$. As base case, note that for tree depth $n = 0$, the claim is trivially satisfied. For any greater tree depth, subsequently consider every child node $v'$ of $r$ in $q$. Assume $l \in L_1$ and $l(\pi(r), \pi(v')) \in E^R_{\sim}$. Then, by definition of the index graph, there must exist $w \in \pi(r)$ and $w' \in \pi(v')$ with $l(w, w') \in E^R$. Now we invoke the induction hypothesis for the subtree $q_{v'}$ of $q$ with root $v'$ (which has then depth $\leq n - 1$). (The backward case follows by symmetry.) Hence, there is a match $h_w$ of the query $q_{v'} \cup \{l(r, v')\}$ into $R$ with $h_w(r) = w$. Yet, by construction of the index graph, $v$ and $w$ are bisimilar, thus there is a match $h_v$ of the query $q_{v'} \cup \{l(r, v')\}$ into $R$ as well. Carrying out this argumentation for every child node of $r$ finally yields a match $h$ for the whole query. □

**Constructing the Parameterized Index** First, a bisimulation on the data graph for the sets of parameters $L_1$ and $L_2$ is needed. For calculating such a bisimulation, we use an adapted version of the algorithm for forward bisimulation equivalence presented in [Fer89] which in turn is an extension of Paige & Tarjan’s algorithm [PT87] for determining the coarsest stable refinement of a partitioning. This algorithm starts with a partition consisting of a single block that contains all nodes from the data graph. This block is successively split into smaller blocks until the partition is stable, i.e., the graph formed by the partition is a complete bisimulation.

In order to create a bisimulation of depth $k$ only, we also compute partitions of the data graph by splitting blocks so that each partition is stable with respect to its predecessor. However, this refinement is performed $k$ times only.

While Paige & Tarjan only dealt with graphs without edge labels, Fernandez extends the algorithm to edge-labeled graphs by applying the refinement operation once for each unique edge label in each step, which results in a time complexity of $O(l|E| \log |V|)$, where $l$ is the number of unique edge labels, $|E|$ is the number of edges and $|V|$ the number of vertices. In order to perform both backward and forward bisimulation
according to the parameters $L_1$ and $L_2$, we essentially exploit the observation that $L_1$-forward-$L_2$-backward bisimulation on a data graph $\mathcal{R} = (V^R, L, E^R)$ coincide with forward bisimulation on an altered data graph $\mathcal{R}_{L_1L_2} := (V^R, L_1 \cup \{l^- \mid l \in L_2\}, E^R_{L_1L_2})$ where $E^R_{L_1L_2} := \{l(x, y) \mid l(x, y) \in E^R, l \in L_1\} \cup \{l^-(y, x) \mid l(x, y) \in E^R, l \in L_2\}$. Therefore, applying the algorithm from [Fer89] to the specified relation sets $L_1$ and $L_2$ yields the desired result, and time complexity $O(|L_1 \cup L_2||E^R| \log |V^R|)$ respectively. Note that $L_1$ and $L_2$ are parameters provided to the algorithm.

After having determined the bisimulation, the resulting blocks from the partition $P$ of the bisimulation are used to form vertices in the index graph. An edge with label $l$ between two blocks (extensions) $E_1, E_2 \in P$ is created for the index graph, if there is at least one pair of vertices $v_1 \in E_1, v_2 \in E_2$ such that the input data graph has an edge with label $l$ from $v_1$ to $v_2$, i.e., $l(v_1, v_2) \in E^R$.

After having determined the bisimulation, the resulting blocks from the stable partition $P$ are used to form vertices in the index graph according to Definition 19.

6.3.5.2. Integrated Query Processing

Recall that the main idea behind structure-aware processing is to apply structure-level processing to obtain a smaller query and smaller sets of candidates, respectively. This strategy makes sense when the effect of this “pruning” outweighs the cost of structured-level processing. Here, we propose an optimization based on the observation that the naive strategy discussed before might fail to perform in the following cases:

- **Simple queries with highly selective triple patterns:** When queries contain only few triple patterns and these patterns have many constants, data-level operations can be performed efficiently. Constants in the patterns can be used to perform index lookup, thus enabling efficient retrieval of data. Also, the amount of data retrieved this way is relatively small and thus, can be joined fast. In this case, there is not much potential gain that can be achieved through pruning.

- **Queries with only a few prunable parts:** This is the case when queries contain many constants and distinguished variables. Structure-level processing only helps to locate candidate groups and data-level processing is still needed for most parts of the query. The
benefit achievable through structure-level processing is thus limited.

- **Queries with high number of index matches:** In this case, the time required for structure-level processing is substantial, thereby canceling out the positive effect of pruning.

**General Idea** Instead of carrying out structure-level and data-level operations subsequently, we perform these operations in an integrated, intertwined way. The approach is based on the following intuitions:

- Data-level operations shall be preferred when processing selective query patterns. In the extreme case when all triple patterns contain constants, the query is processed using the standard approach based on data-level operations only.
- Structure-level operations shall be performed when the prunable part is large. In the extreme case when the entire query can be pruned, i.e., all but root node, it is processed using structure-level operations only as answers can be retrieved directly from the root node.
- Structure-level operations shall leverage results of data-level operations to reduce the number of candidates, and vice versa, i.e., the query is processed via a mixture of structure- and data-level operations. Data-level operations shall be performed for highly selective triple patterns first. The results shall be propagated to the structure-level for pruning index match candidates. In turn, these results are propagated down to the data-level for pruning candidate answers. Thus, the idea of pruning is applied both at the data- and the structure-level.

**Detailed Algorithm** The main procedure for integrated query processing is shown in Alg. 3. Given the query $q$, it outputs a table containing answers to $q$.

Unlike the naive strategy, the optimized procedure starts with identifying prunable parts $q' \in Q'$ when the query consists of more than one triple pattern (line 2). Then, triple patterns $e(v_{q1}, v_{q2})$ are worked off. This is performed in the order of their selectivity (line 5), where selectivity of a triple pattern $e$ is determined using standard statistics derived from the data [SSB+08]. At every iteration, we check if $e(v_{q1}, v_{q2})$ belongs to one of the prunable query parts. If so, structure-level processing will be performed (lines 5-7), data-level processing otherwise (line 12-13).
Algorithm 3: Integrated Structure- and Data-level Query Processing

**Input:** Query graph $q(V^d_{\text{var}}, V^u_{\text{var}}, V_{\text{con}}, L^Q, E^Q)$; data graph $R(V^R, L, E^R)$.

**Data:** Patterns from $E^Q$ are ordered in queue $E^Q_{\text{sorted}}$ according to selectivity; prunable query parts $Q'$.

**Result:** Table $A$ of answers for $q$ on $R$ if $|E^Q| > 1$ then

1. $Q' \leftarrow$ prunableQueryParts($q, n, L_1, L_2$).

2. while $|E^Q_{\text{sorted}}| \neq 0$ do

3.   $e(v_{q1}, v_{q2}) \leftarrow E^Q_{\text{sorted}}$;pop().

4.     if $e(v_{q1}, v_{q2})$ is contained in the triple pattern set $E^Q'$ of some $q' \in Q'$ then

5.       RootInst $\leftarrow$ structureJoin($q', A$).

6.       $A \leftarrow A \boweq \{v_{q1}^\text{root} \rightarrow r \mid r \in \text{RootInst}\}$.

7.       $E^Q_{\text{sorted}}$;remove($E^Q'$).

8.     else

9.       $E^R_e(v_{q1}, v_{q2}) \leftarrow \{v_{q1} \leftrightarrow v_1, v_{q2} \leftrightarrow v_2 \mid e \in E^R\}$.

10.      $A \leftarrow A \boweq E^R_e(v_{q1}, v_{q2})$.

11.    end

12.   end

13. end

Computing prunable query parts is shown in Alg. 4. For the parameters $n, L_1, L_2$, this procedure investigates the query to compute all tree-shaped query parts of $q$ containing undistinguished variables only. This is performed by working off leaf nodes of $q$. Starting from the leaves, the nodes are traversed and labeled (through the function $\lambda$), stopping if (1) one neighbor to be visited is a distinguished variable node or (2) is a root node of a tree structure of depth $n$ or (3) one neighbor edge is not labeled with elements in the sets $L_1, L_2$. In the end, all triple patterns containing only labeled nodes are grouped into connected components which constitute exactly the prunable tree-shaped query parts.

During structure-level processing, prunable query parts $q'$ are processed using structure-level join. This subprocedure shown in Alg. 5 matches $q'$ against the index graph $R_{\sim}$ and outputs a result set $\text{RootInst}$ containing data graph elements that match the root node $v_{q'}^\text{root}$ of $q'$. To restrict the possible instantiations of $v_{q'}^\text{root}$ on the index level, only extensions compatible with the data-level processing performed previously are taken into account. For this purpose, $A_{q'}$ is instantiated (line 2) by taking only extensions in $\text{RootExt}$, i.e., extensions of data elements in the intermedi-
Algorithm 4: Search for Prunable Query Parts

**Input**: Query \( q = (V^d_{\text{var}}, V^u_{\text{var}}, V_{\text{con}}, L^Q, E^Q); n; L_1, L_2. \)

**Data**: Partial labeling function \( \lambda \) assigning natural numbers to query vertices.

**Output**: Set of prunable tree-shaped query parts \( Q' \).

1. \( \lambda(v) \leftarrow 0 \) for all \( v \in V^d_{\text{var}} \) occurring in only one \( e \in E^Q. \)

2. repeat

3. \hspace{1em} foreach \( v \in V^d_{\text{var}} \cup V^u_{\text{var}} \cup V_{\text{con}} \) do

4. \hspace{2em} if all but one neighbor node of \( v \) are in \( V^u_{\text{var}} \) AND labeled with values

5. \hspace{3em} \( \lambda(v) \leftarrow 1 + \max_{\text{neighbors } v'}(\lambda(v')). \)

6. \hspace{1em} end

7. until \( \lambda \) does not change;

8. \( E^Q' \leftarrow \{ e(v_{q1}, v_{q2}) \in E^Q \mid v_{q1} \text{ or } v_{q2} \text{ is labeled} \}. \)

9. \( Q' \leftarrow \{ q' \mid q' \text{ is a connected component from } E^Q' \}. \)

10. return \( Q' \).

ate result \( A \) instantiating \( v_{q}^{\text{root}} \), into account (line 1). Now, starting from

\( v_{q}^{\text{root}} \), all triple patterns of \( q' \) are visited via breadth-first search (line 4). Edges of the structure index which match the triple pattern are retrieved (line 5) and combined with the intermediate results \( A_{q'} \) (line 6). Note that since \( A_{q'} \) has been initialized with extensions of data elements matching the root node (line 2), only structure index edges that can be joined with these elements are kept during this process. Thus, data-level results obtained previously are leveraged to reduce the number of candidates edges during structure-level join.

The results of this structure-level processing feed back into the main query processing procedure. Data elements in the result table \( \text{RootInst} \) are joined with element in \( A \) (line 8). Thereby, only data elements satisfying the query part \( q' \) are kept in \( A \), i.e., structure-level results are propagated back to the data-level join process.

During data-level processing, triples matching the current triple pattern are retrieved from the data graph \( G \) (line 12) and combined with the intermediate results in \( \mathcal{R} \) (line 13).

**Example 21** This example illustrates how the query in Fig. 6.2(c) is processed against the data graph in Fig. 6.2(a).

**Example 21** This example illustrates how the query in Fig. 6.2(c) is processed against the data graph in Fig. 6.2(a).

During the search for prunable parts of \( q \), the following labeling will be obtained: \( v_u, w_u \rightarrow 0, y_u \rightarrow 1 \). Thus we obtain the set of prunable patterns \( E^Q' = \{ \text{supervises}(w_u, x_d), \text{authorOf}(x_d, y_u), \text{conference}(y_u, v_u) \} \).
Algorithm 5: Structure-level Join Processing

**Input:** The index graph $\mathcal{R}_\sim = (V^R_\sim, L, E^R_\sim)$; prunable tree-shaped query part $q'$ with triple pattern set $E^q_\sim$; intermediate result table $A$.

**Data:** Set $\text{RootExt}$ containing candidate extensions for the root $v^\text{root}_q$ of $q'$; Table $A_q'$, where rows represent matches of query part $q'$ onto $\mathcal{R}_\sim$.

**Output:** Set $\text{RootInst}$ of data graph matches for $v^\text{root}_q$.

1. $\text{RootExt} \leftarrow \{ [v]_{\sim} | \mu(v^\text{root}_q) = v \text{ for some } \mu \in R \}$.
2. $R_q' \leftarrow \{ v^\text{root}_q \mapsto v_{\sim} | v_{\sim} \in \text{RootExt} \}$.
3. while $E^q_\sim \neq \emptyset$ do
   4. $e(v_{q1}, v_{q2}) \leftarrow \text{nextBFSNeighbor}()$.
   5. $E_{\sim e(v_{q1}, v_{q2})} \leftarrow \{ v_{q1} \mapsto v_1, v_{q2} \mapsto v_2 | e \in E_\sim \}$.
   6. $A_q' \leftarrow A_q' \bowtie E_{\sim}(e)$.
4. return $\text{RootInst}$.

As they are connected, they form one query part $q'$.

For processing the query, we start with $\text{age}(x_d, 29)$, as $\text{age}$ is the most selective predicate. From the data graph we retrieve all possible bindings for $x_d$: $p1, p3, p5$, and $p6$. When processing the next triple pattern $\text{authorOf}(x_d, y_u)$, we observe that this pattern is part of the prunable part $q'$ which is to be processed on the index level. Hence, we determine $\text{RootExt}$ by looking up what extensions the possible instances of $x_d$ belong to: $E2, E6, E7$. We now retrieve the index level edges for the pattern $\text{authorOf}(x_d, y_u)$: $(E2, E4)$ and $(E6, E4)$. The join with $\text{RootExt}$ does not further reduce this edge set in this case. Next, we process the triple pattern $\text{supervises}(w_u, x_d)$. In the index graph, there is only one $\text{supervises}$ edge: $(E1, E2)$. The join with $\text{RootExt}$ leaves us with only one index graph match: $w_u \mapsto E1, x_d \mapsto E2, y_u \mapsto E4$. Processing $\text{conference}(y_u, v_u)$, the last triple pattern of $q'$, yields $E6$ as the only binding for $v_u$.

Hence, on the structural level, we found $E2$ as match for $x_d$. Returning to the data level, we can rule out the entities $p5$ and $p6$ as bindings for $x_d$. Joining the two remaining matches with the $\text{worksAt}$ edges results in the $(x_d, z_u)$ assignments $(p1, i1)$ and $(p3, i2)$. The second one of these is ruled out by the next triple pattern $\text{name}(z_u, A\text{IFB})$. The two remaining joins on the data level do not change this result. As answer, $p1$ is returned as the only binding to $x_d$. 
6.3.6. Algorithm Analysis

6.3.6.1. Soundness and Completeness

Based on the properties elaborated for the structure index, soundness and completeness of the proposed query processing method are established as follows.

**Proposition 6** Given a data graph, its associated index graph and a query, the result obtained by Algorithm 3 is sound and complete, i.e., it returns exactly all data graph matches.

**Proof Sketch** Soundness can be inferred from the soundness of the join operations at the data graph level for the non-pruned query part together with the second part of Proposition 5 that guarantees soundness of the schema-level processing for the prunable query parts.

Completeness of the retrieved results is guaranteed by completeness of the join operations at the data graph level for the non-prunable query part and by the first part of Proposition 5 ensuring that any match of a query into the data graph (hence also any match of prunable query parts) is “witnessed” by an according match into the index graph. Therefore, constraining the search for prunable query part matches by according index matches will not result in the loss of any match. Finally, the consecutive joining finds all the possible data matches per index match.

6.3.6.2. Complexity

Complexity of standard query processing [AMMH07, WKB08, NW08] is \( O(\text{edgemax} |E^Q|) \), where \( |E^Q| \) denotes the number of triple patterns and edgemax is \( |\{(v_1, v_2) \mid l(v_1, v_2) \in E^R\}| \) with \( l \) being the property instantiated by the largest number of edges, i.e., edgemax denotes the size of the largest table [AMMH07]. This is because joins have to be performed \( |E^Q| \) times and every operation \( A \bowtie E^R \) that join a set of intermediate answers \( A \) and the edges \( E^R \) can be calculated in at most \( |A| \cdot |E^R| \) time and space. This cost cannot be avoided in the general case but in practice, techniques for partitioning and indexing [AMMH07, WKB08] result in near-linear behavior. Compared to this, the complexity of our approach is as follows:

**Proposition 7** For a query with \( |E^Q| \) triple patterns, the time and space for structure-aware query processing is bounded by \( O(\text{edgemaxidx} |E^Q| + \)
\[ \text{edge} \text{max} \text{data}^{E^Q_{\text{pruned}}} \text{ where } \text{edge} \text{max} \text{idx} = \max_{l \in L} |\{l(E_1, E_2) \in E^R\}| \text{ and } \text{edge} \text{max} \text{data} = \max_{E_1, E_2 \in V^R, l \in L} |\{l(v, v') \mid v \in E_1, v' \in E_2\}|. \]

**Proof Sketch** The complexity of our algorithm is composed of the complexity for data-level as well as index-level joins. The cost for computing index matches is \(O(\text{edge} \text{max} \text{idx}^{\mid E^Q \mid})\), where \(\text{edge} \text{max} \text{idx}\) is bounded by the size of the index graph (presisely: the label \(l\) that is associated with the largest number of edges). Data-level joins have to be performed only along the pruned version \(E^Q_{\text{pruned}} \subseteq E^Q\) of the query. So we obtain a complexity of \(O(\text{edge} \text{max} \text{data}^{\mid E^Q_{\text{pruned}} \mid})\), where \(\text{edge} \text{max} \text{data}\) is bounded by the size of the largest extension. □

Compared to existing approaches [AMMH07, WKB08, NW08], less data have to be retrieved from \(G\) (i.e. \(\text{edge} \text{max} \text{data} \leq \text{edge} \text{max} \text{because structure-based data groups are more fine-granular}), and also, fewer joins are required, i.e. \(|E^Q_{\text{pruned}}| \leq |E^Q|\). The overhead introduced to achieve this \(O(\text{edge} \text{max} \text{idx}^{\mid E^Q \mid})\).

6.3.6.3. Structure Index Parametrization

Note that the parametrization has an effect on both the overhead and the gain introduced by structure-based query processing: When more labels (larger depth) are used, the index graph and thus \(\text{edge} \text{max} \text{idx}\) becomes larger. On the other hand, more labels (larger depth) can be considered for query pruning, potentially resulting in larger prunable query parts (i.e., smaller \(|E^Q_{\text{pruned}}|\)). Also, the physical groups obtained from structure-based partitioning become more fine-grained, thereby reducing the size of \(\text{edge} \text{max} \text{data}\).

Thus, it is not straightforward to derive the optimal parameterization for the general case. However, the parameters can be used for fine tuning the approach w.r.t. the given workload. One simple strategy is to support only the labels and the maximum query depth derived from the workload, i.e., \(L_1, L_2\) contains all query predicates and the length of the longest query path is \(\leq n\). More fine-granularity, the parametrization can be done for classes of queries, which are used frequently. In this case, structure-based technique can be seen as an optimization technique, which is applied only for some queries.

Note that while the integrated evaluation strategy does not change the worst-case complexity, it aims at minimizing both \(\text{edge} \text{max} \text{idx}\) and
edgemaxdata by prioritizing cheap operations to quickly filter out irrelevant candidates.

6.3.7. Comparison to Related Work

Related work can be found in three different categories. There are approaches on RDF data management, which have been briefly discussed in the beginning of this chapter. Further, more closely related are the work on structure index that have been investigated for the problem domain of semi-structured and XML data. Different structure indexes have been used for schema-agnostic and for optimized query processing. We have provided a brief discussion on this body of work in Section 6.2. Now, we compare them against the SemSearchPro’s approach for query processing.

6.3.7.1. RDF Data Management

Our solution is complementary to the state of the art concepts proposed for RDF data management. The inverted index design proposed in [DH07] can be used to implement multiple indexes [HD05, WKB08] for supporting different lookup patterns. Specialized indexes [UPS07, LH05] for supporting join patterns might be created. Also, the aspects of index compression and query plan optimization elaborated in [NW08] are orthogonal. Note that as a result of the integrated strategy, only data-level operations might be performed in some cases. Thus, structure-aware query processing can be seen as an optimization of standard query processing [AMMH07], which might apply to only certain cases.

Our solution improves the state of the art in data partitioning by reducing I/O costs. Instead of retrieving data for every single triple pattern using a vertical table, we take the entire query structure into account for the retrieval of candidate data groups. Compared to a vertical table, such a data group is more fine granular, and less likely to contain irrelevant data, i.e., data that satisfy a triple pattern but not the entire query pattern. Also, we improve the state of the art in query processing by combing standard data-level operations with structure-level processing. While [AMMH07] enables efficient join processing, it does not directly solve the proliferation of joins and unions required for answering complex graph shaped queries. Structure-level processing helps to prune the query. In the extreme case where no candidates can be found in the structure index, we can skip data-level processing completely. Finally, through the use of structure index, our solution is also applicable to schema-less data.
We compared our solution with the state of the art in data partitioning and join processing [AMMH07]. Through a benchmark on commonly used datasets, we show that our solution is 7-8 times faster w.r.t. a given query workload, and optimizations result in further efficiency gain.

We now compare our work against the approaches elaborated for the management of semi-structured and XML data presented in Section 6.2.

6.3.7.2. Structure Index

Using a technique similar to the construction of 1-Indexes, we achieve an index size that is also bounded by the size of the data graph. Similar to that work, the strategy for index construction corresponds to minimizing a nondeterministic finite automaton. As noted before, this is different than the strategy for dataguide construction, which resembles the conversion of a nondeterministic finite automaton into an equivalent deterministic automaton.

The main differences between previous work on structure index [BDFS97, KSBG02, QLO03] and our work are as follows: (1) the proposed structure index can be constructed from general graph-structured data as opposed to rooted data graph. Thus, it is also applicable to Web data such as RDF. (2) Its concept of structural similarity is more fine grained, as it rests on characterizing the neighborhood via trees instead of paths. As opposed to the A(k)-Index, which is the most sophisticated index of its kind that can be parameterized based on the depth of paths, the structure index we propose can be parameterized based on both the depth and the labels of tree structures.

6.3.7.3. Optimized Query Processing using Structure Index

Our structure-aware processing technique is similar to work on evaluating path queries using structure indexes such as dataguides [GW97b]. The procedure proposed in [MS99] for instance is similar to structure-level processing. However, since the queries we consider are graph-structured, structure-level processing alone is not sufficient. Our technique involves additional steps such as query pruning and data-level processing.

XML-query processing techniques, especially the ones discussed in Section 6.2 that focus on the task of Twig pattern matching [ZND+01, BKS02, CLL05], rely on tree structures. In particular, they assume that relationships among elements are either parent-child or ancestor-descendant. They are not applicable to our setting, where both query and data are graph structured, and different edge labels have to be considered for matching.
More specifically, the technique employed in our approach is a combination of (a) processing path queries [GW97b, BDFS97, MS99] and (b) twig pattern matching [ZND+01, BKS02, CLL05]. Similar to (a), we match the query against the structure index. However, processing graph-structured queries requires additional examination of the data. Similar to (b), we compute intermediate results which are then joined to obtain the final answers. While intermediate results in (b) are simple root-to-leaf paths of the twig pattern, we retrieve and operate only on data elements that satisfy the structural constraints of the entire query.

The idea of structure-based partitioning is similar to the notion of Prefix Path Streaming employed by [CLL05] for twig pattern matching. However, the index is used here not only for streaming (i.e., for partitioning) but also for query pattern matching. This way, we can guarantee that data elements are only read from disk when they satisfy the structure constraints of the query. The procedure for structure-based processing in fact satisfies the optimality conditions elaborated in [CLL05]. Since index graph matches are grouped together and evaluated as classes, each “stream” of data elements (i.e., elements of an extension) matching a query predicate is scanned only once (Opt1), also, no joins are performed redundantly (Opt2).

6.4. An Empirical Study of Query Processing in SemSearchPro

We now present our evaluation setting and results.

6.4.1. Datasets and Indexes

We use DBLP for the experiments, which captures bibliographic information about the field of Computer Science. Further, we used the data generator proposed for the LUBM benchmark to create 4 datasets for 1, 5, 10, and 50 imaginary universities.

For these datasets, the number of edges are shown in Table 6.2. For comparison, Table 6.2 also contains the number of edges for three different versions of the structure index. The first one called $\mathcal{R}_{\text{Full}}$ is calculated using complete bisimulation. The second one, called $\mathcal{R}_{L_1,L_2}$, represents a label-parameterization that was derived from the the query workload, i.e., by searching for prunable query parts in the given queries and setting $L_1, L_2$ such that labels of all edges contained in these parts are included. The third one denoted $\mathcal{R}_{\sim-d}$ is based on depth-parameterization
with $n = 2$. The results show that while the size of the full indexes $\mathcal{R}_{\text{Full}}$ is almost as big as the size of the dataset (62%-87%), the size of $\mathcal{R}_{\text{Param}}$ is much smaller (4%-30%). The size of the depth-parameterized indexes $\mathcal{R}_{2-d}$ makes up only a small percentage (0.08%-2%).

<table>
<thead>
<tr>
<th></th>
<th>$\mathcal{R}_{2-d}$</th>
<th>$\mathcal{R}_{L1,L2}$</th>
<th>$\mathcal{R}_{\text{Full}}$</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>1278</td>
<td>557,423</td>
<td>11,600,000</td>
<td>12,920,826</td>
</tr>
<tr>
<td>LUBM1</td>
<td>665</td>
<td>30,641</td>
<td>87,590</td>
<td>100,577</td>
</tr>
<tr>
<td>LUBM5</td>
<td>736</td>
<td>151,396</td>
<td>553,362</td>
<td>722,987</td>
</tr>
<tr>
<td>LUBM10</td>
<td>552</td>
<td>253,088</td>
<td>1,127,231</td>
<td>1,272,609</td>
</tr>
<tr>
<td>LUBM50</td>
<td>559</td>
<td>343,682</td>
<td>5,754,191</td>
<td>6,654,596</td>
</tr>
</tbody>
</table>

Table 6.2. Statistics for the data graphs and structure indexes.

### 6.4.2. Queries

Queries in real world scenarios are of different complexities and capture different access patterns. We aim to capture these differences by proposing queries constructed from various combinations of query variables, shape and length. Here, shape refers to the structure of the query graph and length is determined by the longest path in the query graph.

In particular, we have defined 15 conjunctive queries for each dataset, resulting in a total of 30 queries. The 15 queries can be decomposed into five classes, each representing a particular query shape. Each class comprises three queries that vary in length, in the number of variables (constants) and in selectivity.

**Single-atom queries** Single-atom queries consist of exactly one single query atom. Query 1 on DBLP ($Q_{DBLP1}$) for instance, simply asks for all persons. We show all single-atom queries used in the evaluation in Tab. 6.3.

**Path queries** Path queries consist of several connected query atoms that, together, form a path. The example query $Q_{LUBM6}$ retrieves all students and courses that are lectured by full professors. Tab. 6.4 shows all path queries used in the evaluation.

**Star queries** Star queries are composed of more than two single-atom and path queries. These parts share exactly one common node, the center node of the star. The query example $Q_{LUBM12}$ retrieves names of persons, who are both editors and authors, and whose fathers have been cited. Tab. 6.5 shows all star queries used in the evaluation.

**Entity queries** Entity queries constitute a special type of star query that is very common. It is formed by several single-atom queries that share
Table 6.3. Single atom queries for the LUBM and DBLP datasets.

exactly one common node. Intuitively speaking, this node stands for an entity and edges represent entity properties and attributes, respectively. $Q_{DBLP}^9$ asks for entities who have the specified email address, research interest and telephone. We show all entity queries used in the evaluation in Tab. 6.6.

**Graph-structured queries**

Graph-structured queries consist of nodes and edges that form a graph. As opposed to the star query, queries of this type might contain cycles or loops. $Q_{LUBM}^{15}$ for instance, retrieves authors $x$ and authors $a$ that match some specified constraints, i.e. both shall be member of the same organization, a telephone number is specified for the advisor, and the author shall be taking the course run by $FullProfessor^5$ and also, has authored $Publication^7$. We show all graph-structured queries used in the evaluation in Tab. 6.7.

### 6.4.3. Systems

As discussed, the indexing scheme used for YARS and Hexastore [WKB08], the index implementation adopted by Semplore and the index compression and query optimization implemented by RDF-3X [NW08] are orthogonal to our work. Thus, we compare our work with the more related
approach on vertical partitioning [AMMH07], i.e., the state of the art in RDF data partitioning and join processing.

6.4.3.1. Indexing

Aiming at making the systems comparable, we employ the inverted index as a common data structure. This decision follows recent trends in indexing large amount of web data. Conceptually, an inverted index is a list of terms, where each term is associated with a sorted list of documents containing that term. Further, each of that document might be associated with a sorted list of positions indicating where the term appears in the document. We have shown in [WLP+09] have shown that this data structure is a viable choice for indexing triples. For instance, an index called subjOf is proposed, where edge labels of triples are treated as terms, subjects are documents and objects are stored as position information. Given a keyword query like “name”, subjOf can be used to retrieved all triples of the form name(x, y) where every subject x is retrieved from the list of documents associated with the term name, and for each x, objects y are retrieved from the list of positions.
In [DH07], an inverted index has been proposed for indexing data spaces, i.e. sets of triples (like RDF data). The main difference here is the use of concatenated terms in combination with prefix search. In the attribute inverted list (ATIL) for instance, a term is a concatenation of object and attribute name, e.g. KIT//name. Thus, ATIL can be used to retrieved all triples of the form p(x, KIT) given “KIT/*” and the triples of the form name(x, KIT), given “KIT/name”.

Following the design proposed in these approaches, we have implemented indices for the systems under consideration:

**Indexes for Structure-based Partitioning** Both SQP and OSQP rely on structure-based partitioning. For this, two indexes are required, i.e. $R_{\sim}Idx$ for accessing elements of the index graph and $RIdx$ for the data graph. Since $R_{\sim}Idx$ is relatively small, it is kept in memory. $RIdx$ is used to retrieve triples $\langle s, p, o \rangle$ containing elements of a given extension.


\[ Q_{LUBM}^7 \]

\[
\text{SELECT } ?x \text{ WHERE } \\
\begin{align*}
& ?x \text{ rdf:type lubm:FullProfessor} . \\
& ?p \text{ lubm:publicationAuthor } ?x . \\
& ?x \text{ lubm:name } ?n . \\
\end{align*}
\]

\[ Q_{LUBM}^8 \]

\[
\text{SELECT } ?x \text{ WHERE } \\
\begin{align*}
& ?x \text{ rdf:type lubm:Department} . \\
& ?y \text{ lubm:subOrganizationOf } ?x . \\
& ?z \text{ lubm:memberOf } ?x . \\
\end{align*}
\]

\[ Q_{LUBM}^9 \]

\[
\text{SELECT } ?x \text{ WHERE } \\
\begin{align*}
& ?x \text{ lubm:emailAddress} \\
& \text{‘FullProfessor4@Dep8.Uni0.edu’} . \\
& ?x \text{ lubm:researchInterest } \text{‘Research24’} . \\
& ?x \text{ lubm:telephone } \text{‘xxx−xxx−xxxx’} . \\
\end{align*}
\]

\[ Q_{DBLP}^7 \]

\[
\text{SELECT } ?x \ ?a \text{ WHERE } \\
\begin{align*}
& ?x \text{ opus:book_title } \text{‘WWW’} . \\
& ?z \text{ opus:isIncludedIn } ?x . \\
& ?t \text{ opus:isIncludedIn } ?x . \\
& ?x \text{ opus:editor } ?a . \\
\end{align*}
\]

\[ Q_{DBLP}^8 \]

\[
\text{SELECT } ?x \text{ WHERE } \\
\begin{align*}
& ?x \text{ rdf:type foaf:Person} . \\
& ?y \text{ opus:author } ?x . \\
& ?z \text{ opus:editor } ?x . \\
\end{align*}
\]

\[ Q_{DBLP}^9 \]

\[
\text{SELECT } ?x \ ?n \text{ WHERE } \\
\begin{align*}
& ?x \text{ rdf:type foaf:Person} . \\
& ?y \text{ opus:editor } ?x . \\
& ?x \text{ foaf:name } ?n . \\
\end{align*}
\]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Table 6.6.} Entity queries for the LUBM and DBLP datasets. \\
\hline
\end{tabular}
\end{table}

\[ [s] \in R_{\sim Idx} \] as subject. In particular, only those triples that satisfy the query predicate \( p \) (scan) or those that also match the specified constant shall be returned. Since query constants might denote the subject \( s \) or the object \( o \), two separate indexes are employed for indexing triples. The index called extension-property-subject-object EPSO is used to lookup triples with the constant appearing at subject position whereas EPOS is used for lookup with the constant appearing at object position. When constructing EPSO (EPOS), the concatenations of extension id, edge label, subject label (object label), and object label (subject label) are used as terms, subjects (objects) are stored as documents and objects (subjects) are stored as positions. In Fig. 6.6, a generic example is shown for EPOS. Given “E//p//l”, EPSO can be used to find the corresponding \( \text{subject}^{e,p}_1 \), along with the list of associated objects \( \{ \text{object}^{e,p,1}_1, \ldots, \text{object}^{e,p,1}_k \} \) stored in the position list. When only the extension id and property are specified, i.e. “E//p/**”, all \( \{ \text{subject}^{e,p}_1, \ldots, \text{subject}^{e,p}_n \} \) will be returned.

**Indexes for Vertical Partitioning** The main idea behind vertical partitioning is to create two-column tables \( (s,o) \) for every property and store them in a column-oriented database. Two indexes are employed, one for
lookup of subjects in column $s$, given a property and object value, and the 
other for lookup of objects in column $o$, respectively.

Two inverted indexes are employed for these purposes:

- **PSO:** The concatenations of edge label and subject label are used 
as index terms, objects associated with these labels are stored in one 
document
- **POS:** The POS index is created in an analogous manner, but the 
catenations of edge and object labels are used as index terms 
and the documents store subject labels instead of object labels.

In both indexes the objects and subjects, respectively, are sorted to en-
able fast linear merge joins. Subjects and objects are physically stored 
in two separate lists, which corresponds to managing entries of the ta-
bles $(s,o)$ in a column-oriented way. Given “$//p//1$”, PSO can be used to 
find the corresponding $subject^p_1$, along with the list of associated objects 
$\{object^p_1, \ldots, object^p_n\}$ stored in the document. When only the property 
is specified, i.e. the search term is “$//p//*$”, all $\{subject_1^p, \ldots, subject_n^p\}$ 
will be returned.

### 6.4.3.2. Query Processing

For all systems, query predicates are worked off in the same manner us-
ing the same implementation for lookup, scan and join. As a representa-
tive example, we will now discuss how query predicates are evaluated for 
VPQP. This procedure does not depend on any particular evaluation order. 
That is, consecutive query edges (predicates) do not have to be adjacent. 
For our evaluation, the edge order was determined manually (which is the 
same for all systems).

The algorithm takes the current query edge $p(x, y)$ and a list $A$ of re-
Algorithm 6: Evaluating a query predicate

**Input:** query edge \( p(x, y) \), adjacent edge \( p_{next}(v, w) \), a list \( A \) of result tables, \( RIdx \)

**Output:** an updated list \( A \) of result tables

1. Remove the table \( S \) from \( A \) which contains the column corresponding to \( x \).
2. Remove the table \( O \) from \( A \) which contains the column corresponding to \( y \).
3. **if** \( S \) is null and \( O \) is null **then**
   4. **if** \( x \) is a constant **then**
      5. \( J \) ← \( RIdx.retrieve(PSO, p, x) \).
   6. **else if** \( y \) is a constant **then**
      7. \( J \) ← \( RIdx.retrieve(POS, p, y) \).
   8. **else**
      9. Choose index to process \( p_{next}(v, w) \).
   10. **end**
4. **else if** \( S \) is not null and \( O \) is null **then**
5. Create new table \( T \) with columns \( x, y \).
6. **foreach** unique \( s \) in column \( x \) of \( S \) **do**
7. \( T \).append(\( RIdx.retrieve(PSO, p, s) \)).
8. **end**
9. \( J \) ← mergeJoin(\( S, T, x \)).
10. **else if** \( S \) is null and \( O \) is not null **then**
11. Proceed analogously to the previous case.
12. **else**
13. Create new table \( T \) with columns \( x, y \).
14. **foreach** unique \( s \) in column \( x \) of \( S \) **do**
15. \( T \).append(\( RIdx.retrieve(PSO, p, s) \)).
16. **end**
17. **if** \( S \) equals \( O \) **then**
18. \( J \) ← join(\( S, T, (x, y) \)).
19. **else**
20. \( J \) ← mergeJoin(\( S, T, x \)).
21. **end**
22. **end**
23. **end**
24. **end**
25. Add \( J \) to \( A \).
26. Return \( A \).
result tables as arguments. Each table in \( A \) contains columns corresponding to nodes in the query graph. Each node is represented in at most one table. \( A \) is empty at the time the first query predicate shall be evaluated, otherwise not. The algorithm starts with taking two tables from \( A \), i.e. table \( S \) containing the “subject node” \( x \) and table \( O \) containing the “object node” \( y \). If these nodes cannot be found in any table in \( A \), the corresponding tables are set to null. There are now four cases: neither node was found, one of the nodes was found or both nodes were found.

If neither node was found, join processing is not necessary and data retrieved for the predicate can directly be added to the list of result tables. If one of the nodes is a constant that node is used to load the data using the corresponding index PSO or POS, respectively. Otherwise, the next adjacent edge is taken into account for choosing the right index such that entries to be joined in the next step come in sorted fashion, i.e. to enable merge join.

If one of the nodes was found, the entities contained in the column for the node that was found are used to load the data for the current predicate. For instance, if \( x \) was found the table \( S \) will be retrieved. First, a new temporary table \( T \) with two columns \( x, y \) is created to store triples for the current predicate. This table is populated by iterating over all unique subjects \( s \) in column \( x \) of \( S \) and appending the triples retrieved from the index PSO (using the predicate \( p \) and \( s \) as index term). Lastly, tables \( T \) and \( S \) are joined on column \( x \), which results in a new table \( J \).

The last case where both nodes were found involves another distinction. If the current predicate connects two nodes that are part of the same result table, i.e. \( S \) and \( O \) refer to the same table. In this case only one join is necessary. The join attribute spans two columns \( x \) and \( y \). If \( S \) and \( O \) do not refer to the same table, i.e. the query predicate connects to parts of the query graph, two joins are necessary.

After evaluating all predicates of the query graph using this procedure, \( A \) will contain exactly one result table, which comprises of substitutions for all query nodes.

### 6.4.4. Evaluation Setting

We carried out the experiments on a machine with two Intel Xeon Dual Core 2.33 GHz processors and 2GB main memory were allocated to the Java virtual machines used for the experiments. All data and indexes are stored on a Samsung SpinPoint S250 200GB, SATA II. Components of
all systems under consideration have been implemented in Java 5. We have conducted two experiments. One is to compare SQP with the baseline VPQP. The other is to see the effects of optimizations, i.e., SQP vs. OSQP to see the effect of the integrated strategies, and how these two systems perform w.r.t. different depth-parameterizations. All times presented represent the average of 10 runs. Between queries, we explicitly clear the operating system cache and the internal caches used by Lucene.

6.4.5. Structure-based Approach vs. Baseline

Through this experiment, we aim to compare SQP with VPQP and to identify the main drivers for the performance differences. Similar to the previous evaluation of triple stores [AMMH07, WKB08], we manually created queries, i.e., 3 queries for each query types, resulting in 15 queries for each dataset and a total of 30 queries (as discussed previously in the subsection on queries).

In Fig. 6.7(a+b), total query processing time is plotted for both approaches. Clearly, SQP is faster than VPQP. For DBLP in particular, SQP performs a factor of 7-8 faster. Further, we note that SQP exhibits much better performance than VPQP w.r.t. queries that have more complex structures, i.e., the queries $Q4$-$Q15$. SQP is slightly worse w.r.t. simple queries, i.e., the single triple patterns $Q1$-$Q3$. This suggests that with more complex queries, the overhead incurred by the additional structure-level processing can be outweighed by the accumulated gain.

To better understand the reasons for this, we decomposed total processing time into the fragment required for structure-level processing, i.e., index matching (match), to retrieve data from disk (load) and to combine them (join). In particular, we compared the time SQP needed for match with the difference for load and join between SQP and VPQP. This is shown in Fig. 6.7(c+d) to illustrate when the additional cost of index matching is outweighed by the gain in loading and join performance. Fig. 6.7(c+d) also illustrates the impact of query pruning measured in terms of the number of query nodes, which could have been discarded after structure-level processing. Along these factors, we will now discuss results for each specific query type.

$Q1$-$Q3$: Single-triple-patterns Here, SQP is slower than VPQP. This is because only single index lookup or table scan is needed to answer these queries. In the cases of index lookup only, i.e., $Q1$-$Q3$ on DBLP and $Q1,Q3$ on LUBM, our approach does not provide any advantages
but requires additional effort for structure-level processing. For Q2 on LUBM, which requires a scan to retrieve all triples matching a predicate, one can see that SQP improves loading time. Performance of SQP equals VPQP in this case, as the gain fully compensates the cost for structure-level processing.

**Q4-Q6: Path Queries** For processing these ones, triples retrieved for the patterns have to be joined along the path. Here, we observe that SQP largely reduces time for load and join w.r.t. all queries. This is especially true for queries containing patterns with low selectivity. E.g. for \(Q_{LUBM6}\) the number of triples retrieved for \(\langle y \in B, teacherOf z \rangle\) is many times smaller than \(\langle y teacherOf z \rangle\), thus essentially improving load and join performance. Another advantage is that path queries lend themselves to be pruned: on average 1.6 query nodes have been removed.

**Q7-Q9: Entity Queries** For these ones, triples need to be retrieved and joined on the node representing the entity. Also here, SQP improves I/O and join performance. SQP outperforms VPQP w.r.t. all queries except for \(Q_{LUBM9}\). We observe that best performance is achieved with queries containing few constants \((Q_{LUBM8}, Q_{DBLP8}, Q_{DBLP9})\). Query patterns without constants can be removed in most cases, thus avoiding unnecessary loads and joins. For instance \(Q_{LUBM8}\) contains only one constant, while \(Q_{LUBM9}\) has 3 constants. Both contain the same number of triple patterns but \(Q_{LUBM8}\) benefits much more from reduction in I/O and join. In fact, the performance gain for \(Q_{LUBM9}\) does not outweigh the cost for structure-level processing. SQP is slower than VPQP in this case, as many
constants can be effectively used for the lookup of triples. There is thus not much data that can be “filtered” through structure-level processing. One could say that the number of constants and the selectivity of triple patterns have an adverse effect on the gain that can be achieved with SQP.

**Q10-Q12: Star Queries** These queries behave similar to entity queries. However, pruning is more delicate as distinguished variables might be not at the center but anywhere. However, since star queries are larger than entity queries on average, larger portion of the queries lend themselves to be pruned. Also for this type of queries, results show that performance of SQP is superior to VPQP. In only one case (with $Q_{LUBM11}$) processing time is the same for SQP and VPQP. It seems that this query is particularly hard to match against the index graph. The index graph contains many similar structures that match the query, resulting in a large number of index matches here.

**Q13-Q15: Graph-shaped Queries** We expected that the relative performance of SQP is best here, since these queries have most complex structures. However, while the performance of SQP is still better than VPQP for all graph-shaped queries, except for $Q_{LUBM15}$, the relative improvement of SQP over VPQP is not as high as the improvement achieved for entity and star queries, i.e., 3-4 times compared to 9-10 times improvement. Results show that the main reason for this is because structure-level processing cost largely increases with complexity.

**Scalability** We measured the average performance for LUBM with varying size, i.e., LUBM for different number of universities (1, 5, 10, 20, 50). Fig. 6.8 shows the differences (VPQP-SQP) in total processing time, load and join. The relative performance of SQP improves with the size of the data. In particular, the differences in performance for load and join increase in larger proportion (the gain) than the overhead incurred for index match. This is because match performance is determined by the size of the index graph. This size depends on the structures in the data but not the actual size of the data. For instance, the fact that the size of $R_{2-d}$ for LUBM10 is smaller than for LUBM5 (see Table 6.2) simply tells that LUBM5 exhibits higher structural complexity (when only connections of length 2 are considered). Thus, the match time does not necessarily increase when the data graph becomes larger. The positive effect of data filtering and query pruning (load and join) however, correlates with the data size.
6.4.6. The Results of Optimization

The last experiment shows that SQP outperforms the state of the art w.r.t. complex queries, especially on large datasets that exhibit homogeneous structures (i.e., have relatively small structure index). In real life scenarios, the dataset itself might increase in size but its structure might remain relatively stable. In this case, SQP scales much better as the gain proportionally increases, while the cost of structure-level processing remains stable. However, the structure index might be too large for datasets that have diverse structures. Also, SQP does not perform well on queries that are simple and (or) do not lend themselves to query pruning.

We will discuss the effects of the two optimizations, which aim to address these problems. While the previous experiment follows the tradition of benchmarking triple store by means of manually crafted queries, the analysis here is based on a larger scale experiment. In particular, we use a query generation procedure to obtain a larger number of test queries (see [TL10] for additional details). It is run for each dataset to obtain 160 queries of different types.

The Effect of the Integrated Strategy

To investigate the combination of structure- and data-level processing, we compare SQP with OSQP. Fig. 6.9 shows the average time for the 160 queries. Clearly, OSQP consistently outperforms SQP. The average time obtained for OSQP is lower than SQP for all datasets, up to 50 percent in the case of LUBM5. This suggests that the integrated strategy is superior.

We provide a breakdown of the average time to understand the effect of query structure. In Fig. 6.10, we show the average time for the different query types, ranging from single triple patterns to more complex graph-shaped queries. For both approaches, query processing time increases with the complexity of the query. For single triple patterns, OSQP slightly improves SQP. In fact, it equals VPQP because in this case, only data-level
operations are performed. The largest difference in performance can be observed for entity queries. These simple queries contain selective patterns and the amount of prunable parts is relatively small. In this case, OSQP prioritize data-level processing to work off the “cheap” patterns first and then, use their results for pruning both structure- and data-level candidates. Improvement can be observed also for more complex structures, i.e., graph-shaped queries.

Figure 6.9. Average SQP and OSQP times for all queries.

The Effect of Bisimulation Depth Fig. 6.11 shows the ratio of structure index size to data size for \( n = 1, 2, 3, \text{full} \). The lines for the LUBM datasets overlap. The line for DBLP is different but also indicates that size largely increases with bismulation depth.

To understand the effect of this parameter on query processing, we run OSQP on several structure indexes with different depth. Fig. 6.12 shows the average time for \( n = 1, 2, 3 \). It increases with the bismulation depth. For greater depth, the structure index size increases, thus resulting in higher overhead for evaluating structure-level joins. While the prunable parts become larger, it seems that the effect of pruning cannot compensate the increased overhead. For the datasets in the experiments, using struc-
Figure 6.11. Ratio of structure index size to data size.

ture index with \( n = 1 \) achieves best performance.

Figure 6.12. Times for different bisimulation depths.

6.5. Conclusions

We have provided a brief survey on the body of work in the problem domains of semi-structured and XML data management. Especially for semi-structured data, the concept of structure index have gained acceptance. It is the principle mean for browsing and for enabling query processing on data for which there exists no explicit schema. We have coined this kind of processing that rely on building a pseudo-schema from schema-less data, schema-agnostic query processing. Dealing with schema-less data is only one aspect. It has been shown that the concept of structure index can be leveraged for more optimized query processing, especially for the tasks of path- and twig-pattern matching.

Based on this existing work, we presented SemSearchPro’s approach to query processing, which essentially amounts to the task of graph-pattern matching. For this, SemSearchPro employs a structure index that can be
built for general graph-structured data. Using this index, schema-agnostic query processing can be supported for any kinds of graph-structured data on the Web, including semantic data such as RDF. Further, the index is also leveraged for more optimized processing of query graph patterns. A procedure coined structure-aware query processing has been introduced. It is based on using a parameterized structure index. The basic idea of structure-aware processing is to match the query against the structure index first, a step called structure-level query processing. This is to prune away certain parts of the queries and to focus on certain parts of the data that more likely contribute to the final results. SemSearchPro’s approach integrates this structure-level processing with standard query processing (coined data-level query processing) to deal with complex graph patterns. This integration is one optimization of the structure-aware technique. The second optimization is based on parameterizing the index, which helps to reduce its size.

In the experiments, it has been verified that this structure-aware approach effectively reduces I/O and the number of joins. Compared to the state of the art, it achieves 7-8 times faster performance w.r.t. a given workload. The gain in performance is evident especially for complex query patterns.
Table 6.7. Graph-structured queries for LUBM and DBLP datasets.
Chapter 7

Conclusions

This book focuses on the subject of Semantic Web Search. To introduce the readers to this new concept, we provided a general discussion on search concepts and clarified the differences between Semantic Web Search and the existing body of work. In particular, we presented a general model for search, which is further divided into the category of document retrieval and data retrieval. Then, we gave a definition for Semantic Search, making clear that there is no precise notion but rather, Semantic Search shall be conceived as a kind of search that employs a semantic model. This is a general definition of Semantic Search that fits to the various Semantic Search solutions that have been developed by researchers and practitioners from different communities. To understand the different viewpoints on Semantic Search that have emerged in different communities, we looked at Semantic Search from the perspectives of the DB, the IR and the Semantic Web communities. Based on the defined search concepts, we introduced Semantic Web Search as a special kind of Semantic Search that is focused on the retrieval of semantic data. While Semantic Web Search is centered on data retrieval, it also supports document retrieval. For this, we showed how to specify rich document models as semantic metadata. Last but not least, we argued for a process-oriented view on Semantic Search. Semantic Search gives the power to address more complex information needs that ultimately, involve complex queries and results. An effective Semantic Search solution must take the resulting complexity facing the user into account. To accommodate this, we proposed to look at the entire process of search, from query construction, to query processing up to result presentation and refinement, and to investigate where in this process semantics can play a role and how it can help
to improve the overall "search process experience".

We surveyed the state of the art of Semantic Web Search, discussing the specific techniques employed for crawling, storing and indexing as well as for querying and ranking semantic data. For this, we studied different lines of work, from approaches that leverage IR technologies, to those that are built upon database concepts up to native ones that are specifically designed and optimized for semantic data management. Besides, the multi-source nature of Semantic Web Search occupied our attention. For retrieving and integrating results from multiple sources, we looked at work on federated query processing and data integration.

For demonstrating the concept of Process-oriented Semantic Web Search, we presented one approach called SemSearchPro. It is basically a compilation of work that integrates different research contributions we had in the past years. In particular, SemSearchPro uses an inverted index for managing and indexing semantic data. Semantic data is transformed into fields and terms of documents, which are then managed using an off-the-shelf IR engine. For multi-source search, SemSearchPro addresses the problem of combining results from multiple sources. More specifically, complementary information from different sources that refer to the same entity must be stitched together. For this, similarities between entities would have to be computed online. To avoid the cost of online computation, SemSearchPro leverages the work on data integration to precompute mappings between entities and to store them in a specialized index. These mappings are then leveraged during online query processing to improve the performance of multi-source search. Besides these concepts, SemSearchPro implements the approaches we proposed for query translation and query processing.

Integrating these approaches, SemSearchPro shows that semantics can be used throughout the process. In particular, it demonstrates that semantics can be used to improve efficiency and scalability. The semantics used by SemSearchPro is captured by a lightweight semantic model that can be computed from the data. Semantics in this sense, simply reflects knowledge about group of entities and relations between them. This knowledge is treated as a summary model of the data. Depending on how much of the data shall be reflected in the summary, we can obtain models of different sizes. For query processing, we use a summary model to guide the process of data partitioning. The resulting strategy called structure-based partitioning basically, groups together data elements that are similar in structure. Leveraging data grouped this way, we proposed a structure-aware
query processing strategy, which can use the more compact summary to validate the structure constraints of the query and to locate groups that satisfy these constraints. Also for query translation, the semantic model acts as a summary model. Performing the translation mainly on this more compact model – instead of using the actual data – also substantially improves scalability and efficiency. Moreover, since the model is derived from the data and is actually employed as a kind of schema, SemSearchPro in fact represents a schema-agnostic approach that does not require the existence of a schema. This is quite important for Semantic Web Search as frequently, semantic data is not accompanied by a schema.

SemSearchPro not only addresses the complexity of processing but also considers the complexity facing the user. It uses semantics to support the user throughout the process. Instead of formulating the needs in terms of formal structured queries, the user can use keyword queries. SemSearchPro can translate these queries to structured queries and present them to the user in abstract syntax, natural language or visualize them as graphs. Upon the selection of the query corresponding to the intended information need, SemSearchPro delivers the results using customized widgets. Depending on the types of results and interactions, appropriate widgets are automatically selected. Refinement of queries and results are possible via operations on facets, which are automatically derived from the semantic model for the current set of results.

We come to the conclusion that there is yet no solution but rather, a range of concepts and technologies have proven to be viable for Semantic Web Search. IR-based solutions can deal with a large amount of data. For simple keyword queries, these solutions can scale to a large Web of semantic data. At least, the experience and knowledge of IR researchers and practitioners who are used to tackle problems at Web scale, can help to deal with the volume of data on the Semantic Web. Harnessing the true potential of Semantic Web Search to support complex information needs however, requires the capability to deal with complex queries and heterogeneous results possibly coming from different sources. Thus besides data volume, we face the challenges of data complexity and heterogeneity. DB-based solutions including specialized data partitioning techniques, indexing strategies, and procedures for query planning and join execution, have been used to support complex query processing. Also the body of work on database integration is valuable for dealing with data heterogeneity. Yet, DB-based solutions do not scale beyond the realm of enterprises. Supporting retrieval on Terrabytes of data or more requires extensive cre-
ation of redundant indexes, putting a large burden on maintenance. In fact, it is an open question whether DB-based solutions can scale to the Web of semantic data.

From the study of the state of the art, the contributions we made, and the experiences we acquired with SemSearchPro, it seems that the consequent adoption of DB concept for data management and query processing combined with the approximate nature of IR techniques reflected in concepts for ranking and top-k processing, constitutes one viable direction towards Semantic Web Search. Further, a process-oriented approach is essential for enabling the user to unfold the power of Semantic Web Search.
Bibliography


[MWL⁺08] Li Ma, Chen Wang, Jing Lu, Feng Cao, Yue Pan, and Yong Yu. Effective and efficient semantic web data management over db2. In *SIGMOD Conference*, pages 1183–1194, 2008.


Index

algebraic document retrieval model, 39
answer space, 109
backward search, 153
basic graph pattern, 52
Basic Semantic Search model, 53
bidirectional search, 153
bisimulation, 102, 188
boolean document retrieval model, 39
clustering, 101
categorical model, 45
conjunctive queries, 51
data access, 37
data integration, 84
data model, 43
data partitioning, 74, 185
data retrieval, 36, 37
data-level integration, 85
dataguide, 181
DB-based semantic data management, 74
document model, 43
document retrieval, 36, 38
embedded semantic data, 50
entity search, 94
faceted search, 131, 135
fact search, 95
federated query processing, 81
form-based search, 131
graph exploration, 152
graph pattern, 52
graph pattern matching, 52
graphical query interfaces, 131
information retrieval, 36
IR-based semantic data management, 71
join implementation, 38
join processing, 38, 75, 186
keyword matching, 151
keyword matching elements, 105
keyword matching score, 150
keyword queries, 51
keyword search, 23, 131, 133, 135
keyword translation, 80
keyword-driven schema-agnostic search, 130
language modeling, 40
linguistic model, 45
linked data, 47
linked data principles, 47
Linked Open Data, 23, 47
mappings, 83
matching framework, 38
metadata, 41
native semantic data management, 76
natural language interfaces, 131
ontologies, 23, 47
ontology-based document model, 44
ontology-based document retrieval, 44
parameterized structure index, 194
path length score, 149
path queries, 182
popularity score, 149
presentation-query model, 57
presentation-resource model, 57
probabilistic relevance framework, 40
Process-Oriented Semantic Search, 58
Process-Oriented Semantic Search model, 56
Process-oriented Semantic Web Search, 60, 89
property-based partitioning, 75
pseudo-schema construction, 100
query completion, 132, 136
query construction, 104, 127
query model, 38, 53
query optimization, 38
query planning, 82
query presentation, 113
query processing, 108, 179, 186
query refinement, 114, 127
query search space, 105
query space, 105, 151
query translation, 141, 150
query-dependent ranking, 78
query-independent ranking, 77
ranking, 148
RDF Schema, 22
RDF triples, 23
RDFa, 50
relation search, 95
Resource Description Framework, 22
resource model, 38, 53
result completion, 132, 136
result presentation, 111
schema-agnostic, 89, 139, 180
schema-level integration, 85
scoring, 148
search, 34
semantic data, 24, 46
semantic data crawling, 68
semantic data indexing, 76, 185
semantic data management, 71
semantic data model, 45
semantic data querying, 71
semantic data ranking, 77
semantic matching framework, 53
semantic model, 45, 53
semantic presentation framework, 58
Semantic Search, 36, 53
semantic translation framework, 58
Semantic Web, 22
Semantic Web data retrieval, 79
Semantic Web languages, 22
Semantic Web Search, 59, 63, 79
Semantic Web Search engines, 63, 79
Semantic Web Search model, 59
semantics, 45
SPARQL, 52
structure index, 181, 188, 194
structure-based partitioning, 75
structure-based query processing, 188
structured data, 36, 40
structured-based partitioning, 189
system-query model, 57
system-resource model, 57
text retrieval, 36
textual data, 40
triple pattern, 52
twig patterns, 183
user-query model, 57
vertical partitioning, 75
Web Ontology Language, 22