Umutcan Serles
Dieter Fensel

An Introduction to Knowledge Graphs

Springer

An Introduction to Knowledge Graphs

Umutcan Serles • Dieter Fensel

An Introduction to Knowledge Graphs

Springer

Umutcan Serles
Department of Computer Science
Semantic Technology Institute, University of Innsbruck
Innsbruck, Austria

Dieter Fensel
Department of Computer Science
Semantic Technology Institute, University of Innsbruck
Innsbruck, Austria

ISBN 978-3-031-45255-0 ISBN 978-3-031-45256-7 (eBook)
https://doi.org/10.1007/978-3-031-45256-7

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Foreword

The standard narrative about the recent history of AI is that a combination of rapidly growing compute power and an immense increase in available data have caused a scaling explosion in AI that has led to the results in Machine Learning in the past decade. Indeed, this narrative is true, and the results of the ML explosion are clear for all to see, in the popular press, in scientific publications, and in real world applications, ranging from product recommendation to fraud detection and from chatbots to face recognition.

What is rather less known to the general public, the popular press, and indeed in AI itself is that a similar scaling explosion has taken place in another area of AI. By the end of the 1990s, a knowledge base of a few thousand facts and rules was considered large. But nowadays, we routinely manipulate knowledge bases that contain billions of facts, describing hundreds of millions of entities, using an ontology of many thousands of statements.

The main driver for this "other explosion" of size in AI has been the adoption of the knowledge graph model, combined with ontology languages that carefully balance expressivity against computational cost. Knowledge graphs now form the biggest knowledge bases ever built, and without a doubt, languages like RDF Schema and OWL are by far the most widely used knowledge representation languages in the history of AI.

And these knowledge graphs have come of age. They are used in science, in public administration, in cultural heritage, in healthcare, and in a broad range of industries, ranging from manufacturing to financial services and from search engines to pharmaceuticals.

It is important to realize that this explosion in the size of symbolic representations did not come out of the blue. It stands in a long tradition of research into symbolic representations and their corresponding calculi. This long tradition includes rule-based languages, non-monotonic logics, temporal logics, epistemic logics, Bayesian networks, causal networks, constraint programming, logic programming, the event calculus, and many many others. So, what was it about knowledge graphs and

ontologies that made them so successful, as compared to all the others? Let me put forward three reasons.

Facts matter. The (implicit) assumption in pretty much all of KR has been that what matters most are the universally valid sentences, the "rules" that describe how the world works: "If t_1 is before t_2, and t_2 is before t_3, then t_1 is before t_3," "All birds fly," "If patient *X* suffers from disease *Y*, and *Y* is a viral infection then *X* will have a fever," "Every country has precisely one capital," etc. But the Semantic Web research program showed us that actually the facts matter perhaps even more. Much of intelligent behavior does not arise out of universally quantified sentences ("Every country has precisely one capital."), but rather from ground predicates: "Paris is the capital of France," "COVID is a viral infection," etc. It is not (or: not only) the inferences we can draw that enable intelligent behavior, it is also (or: mainly) the huge amount of rather mundane facts that we know about the world that allows us to navigate that world and solve problems in it. Or to quote from the introduction of this book: "How useful is an intelligent assistant, if it does not know about the address of the restaurant you are looking for?". This came as rather a shock to KR researchers (and maybe a bit of a disappointment), but it was one of the lessons that Semantic Web research taught us.

Most facts are binary. A further insight (and perhaps further disappointment to KR researchers) was that most facts are simple binary relations. Yes, of course, n-ary relations with $n>2$ exist, and they are sometimes needed, but by far the large majority of facts have the form of "triples": *<object1, hasRelationTo, object2>*. Together, these two insights say that a large volume of binary ground predicates are a crucial ingredient for successful KR, and together they directly lead to knowledge graphs as a natural knowledge model.

"A little semantics goes a long way." (In the immortal words of Jim Hendler). Whereas the instinct of KR researchers had always been to ask, "what is the maximum amount of expressivity I can get before the computational costs become unacceptable," a more interesting question turned out to be "what is the minimum amount of expressivity that I actually need?" And the answer to this question turned out to be "surprisingly little!" RDF Schema, by far the most used KR language in the history of AI, only allows for monotonic type inheritance, simple type inference based on domain and range declarations, and very limited forms of quantification. No non-monotonicity, no uncertainty, not even negation or disjunction. And of course, there are many use cases where some or all of these would be useful or even necessary, but the surprise was an 80/20 trade-off (or even a 99/1 trade-off, who can say) between the large volume of simple things we want to say and the small volume of remaining complicated things.

Among all the other books on knowledge graphs (on formal foundations, on methodology, on tooling, on use-cases), this book does an admirable job of placing knowledge graphs in the wider context of the research that they emerged from, by giving a kaleidoscopic overview of the wide variety of fields that have had a direct or indirect influence on the development of knowledge graphs as a widely adopted storage and retrieval model, ranging from *AI* to *Information Retrieval* and from the *World Wide Web* to *Databases*. That approach makes it clear to the reader that

knowledge graphs were not just some invention "out of the blue," but that instead they stand in a long research tradition, and they make specific choices on the conceptual, the epistemological, and the logical level. And it is this set of choices that is ultimately the reason for the success and wide adoption of knowledge graphs.

Vrije Universiteit Amsterdam,
Amsterdam, The Netherlands
April 2023

Frank van Harmelen

Preface

The overall goal of this book is to give a deep understanding of various aspects of knowledge graphs:

- Fundamental technologies that inspired the emergence of knowledge graphs
- Semantics and logical foundations and
- A methodology to create and maintain knowledge graphs as a valuable resource for intelligent applications

Knowledge graphs[1] can provide significant support for application types such as:

- Applications are being made more ubiquitous every day. Who does not own a smartphone with Cortana, Google Assistant, or Siri?
- Applications are only as powerful as their knowledge. How helpful is an intelligent assistant if it recognizes your speech perfectly but does not know the address of the restaurant you are looking for?

Knowledge graphs, as a flexible way of integrating knowledge from heterogenous sources, will power these intelligent applications.

How to build knowledge graphs? We will cover the following major topics:

- Introduction: motivation, related fields, definition, application types
- The holy grail: machine-understandable semantics
- The underlying base for this: logic
- Set up a knowledge graph: knowledge creation (static, dynamic, and active data)
- How to store a knowledge graph: knowledge hosting
- How good is your knowledge graph: knowledge assessment
- How to fix your buggy knowledge graph: knowledge cleaning

[1] Bonatti et al. (2019), Chen et al. (2016), Croitoru et al. (2018), d'Amato and Theobald (2018), Ehrig et al. (2015), Fensel et al. (2005, 2020), Hogan et al. (2021), Li et al. (2017), Pan et al. (2017a, b), Van Erp et al. (2017).

- How to extend the coverage of your knowledge graph: knowledge enrichment
- How to turn knowledge graphs into applications: knowledge deployment

Knowledge graph technology is a continuation of the *Semantic Web* initiative (see Berners-Lee et al. (2001), Fensel et al. (2005)). Here, for the first time, vast amounts of content and data were semantically annotated on a worldwide scale. Therefore, introducing knowledge graph technology is incomplete without referring to the underlying principles and techniques from the Semantic Web. Besides discussing other related fields, the Semantic Web will be a designated focus of our book. The Semantic Web was invented for two main reasons (Fensel and Musen 2001):

- *The traditional web of documents got too big and heterogeneous to be consumed by humans in a scalable manner.* However, if the Web has machine-understandable semantics, then automated agents and search engines can consume it on behalf of humans.
- *Solving the knowledge acquisition bottleneck:* Artificial Intelligence has failed initially because people realized that it is hard and expensive to acquire knowledge in a form that intelligent applications can understand. Semantic Web crowdsources the knowledge acquisition task to billions of users through machine-understandable annotations.

Therefore, we aim to provide a deep understanding of the Semantic Web and semantic technologies. We introduce:

- The limitations of the traditional Web
- The architecture and principles of the Web and the Semantic Web
- The stack of technologies enabling the Semantic Web
- Semantic Web of content, data, and services and
- Semantic Web applications

Since the Semantic Web is based on other research fields, we will also provide an insight and overview of those research areas that directly lie under the Semantic Web or knowledge graph technology.

During the following, we will shortly introduce the need for knowledge graphs to enable the age of intelligent agents:

- Intelligent search and query answering (Fig. 1)
- Intelligent personal assistants (e.g., Amazon Alexa,[2] Apple Siri,[3] Google Assistant, Microsoft Cortana) and
- Autonomous agents (Fig. 2)

Intelligent Search and Query Answering. Google started in the late 1990s as a search engine for the Web.[4] They were actually a bit late in this but quickly managed

[2] https://alexa.amazon.com/

[3] https://www.apple.com/siri/

[4] https://en.wikipedia.org/wiki/History_of_Google

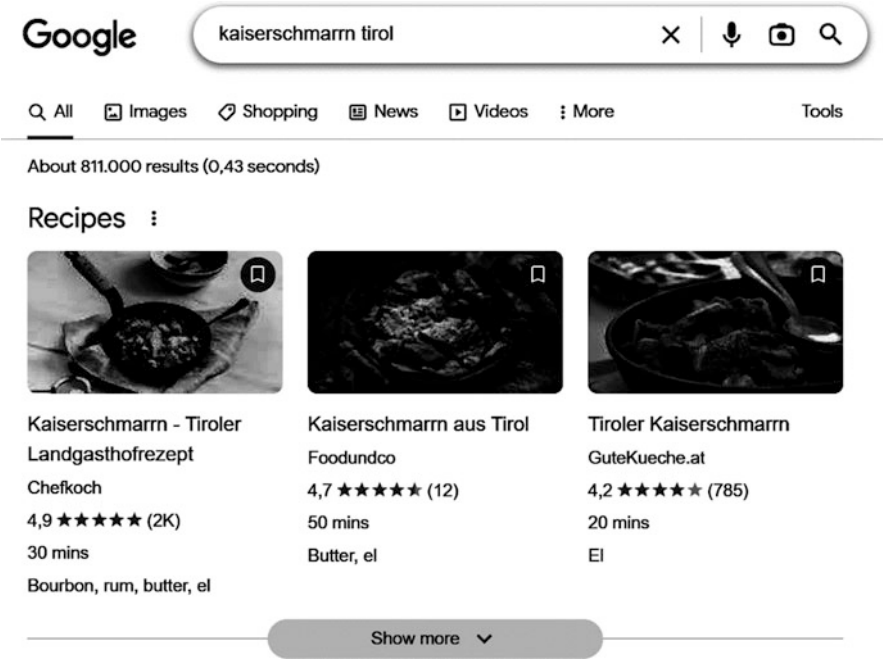


Fig. 1 Intelligent search engines answering with rich snippets

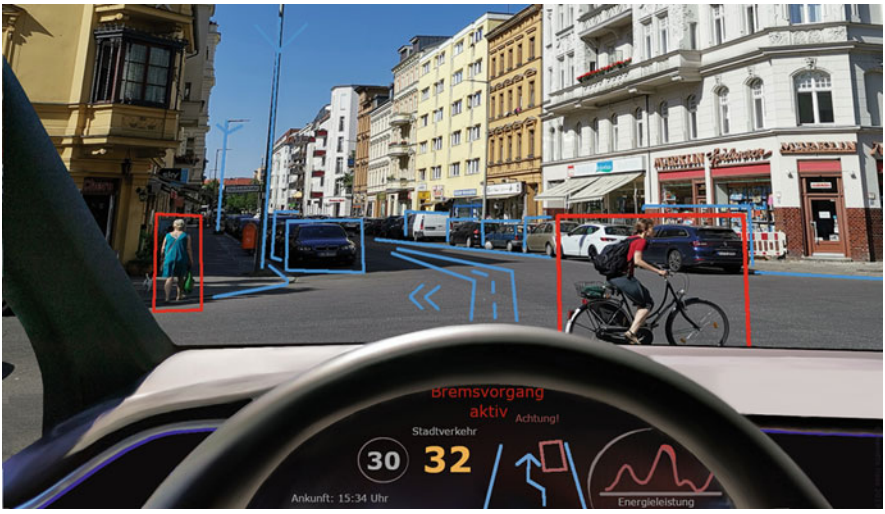


Fig. 2 Autonomous agents (Image by Eschenzweig. Licensed under CC-BY-SA. https://commons.wikimedia.org/wiki/File:Autonomous-driving-Barcelona.jpg)



Fig. 3 Google as a search engine

to outrun their competitors. Now search on the Web is called "googling" on the Web. Their original approach was based on the PageRank algorithm (Page et al. 1999) (see later Chap. 3 on information retrieval), which helped them to show the relevant links first. This made them very successful and turned them into an opponent of semantic technologies. You simply did not need them to provide a proper Web search (see Fig. 3).

However, there was a specific limitation for the advertisement business model attached to it. You find interesting links for the user, and then they leave you the next moment (and also have to extract the wanted information manually from that Web site). What if you try to provide not only valuable links to the users but extract the information from that source and provide them to the users as potential answers (Fig. 4)? You increase the interaction with them; they are not leaving your Web page and you can even try to begin e-commerce with them. For example, when searching for a specific hotel, you could offer them directly a booking possibility. This turned Google from a "simple" search engine into a *query-answering engine*. Suddenly semantics and semantic annotations of the Web became a strategic issue, as without understanding the content on a Web site but only its overall score in PageRank was no longer enough. Instead, as more Web site providers became willing to annotate their Web pages semantically, the more this new approach could work and scale.



Fig. 4 Google as an *intelligent* search engine

Bots are virtual agents that interact with human users, searching for and integrating information on the Web and in other sources on behalf of a human user. Alexa, Siri, and Google Assistant are examples of this. Their natural language understanding capabilities are impressive and based on Big Data and statistical analysis. Also, they can respond in natural language quite well.

Still, there is a severe bottleneck. For example, how can a bot know which restaurants are out there (see Fig. 5)? For a helpful answer, it must know the restaurants, their menus, their opening times, etc. Even the most recent applications like ChatGPT that are trained on a significant portion of the Web may have trouble answering questions accurately. Without such knowledge, all NLP understanding is of little use.

Autonomous driving In March 2018, Elaine Herzberg was the first victim of a fully autonomously driving car.[5] Besides many software bugs, a core issue was that the car assumed that pedestrians cross streets only on crosswalks. Obviously, such assumptions should have been made explicit and confronted with world knowledge captured by a knowledge graph. In that case, she still would be alive!

[5] https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg

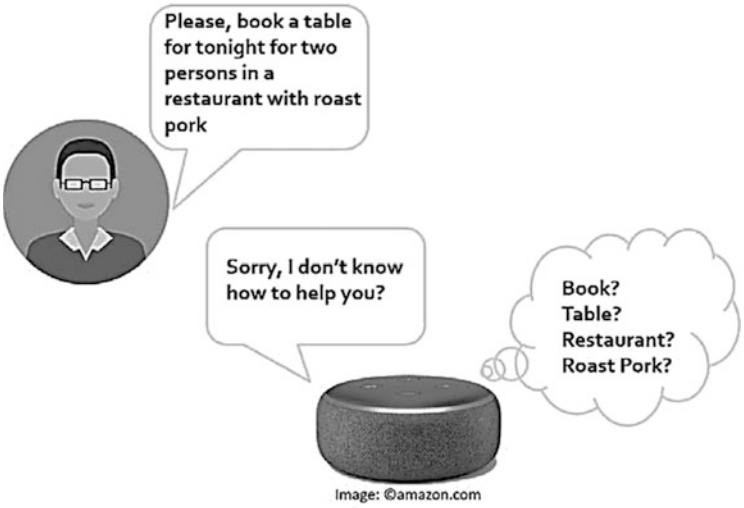


Fig. 5 A bot without world knowledge (The human depiction is taken from https://www.flaticon.com/free-icon/teacher_194935. Licensed with Flaticon license)

What are the conclusions from the requirements of these fields? Explicit knowledge is crucial for intelligent agents to help users to achieve certain goals. Statistical methods can bring us a long way, e.g., text and voice processing is becoming mainstream thanks to the advances in Machine Learning. However, more than just multiplying vectors and matrices are required. Intelligent personal assistants are limited by their lack of knowledge. Autonomous cars may avoid killing people if they have more explicit knowledge about their environments and common sense.

Knowledge graphs are the most recent answer to the challenge of providing explicit knowledge. They contain entities' references and their relationships. They integrate heterogeneous sources and may contain billions and higher numbers of facts.

Summary Our book is about how to build knowledge graphs and make them useful resources for intelligent applications. We focus on the following aspects:

- Part I will provide the overall context of knowledge graph technology.
- Part II will provide a deep understanding of logic-based semantics as the technical core of knowledge graph technology.
- Part III focuses on the building process of knowledge graphs. We focus on the phases of knowledge generation, knowledge hosting, knowledge assessment, knowledge cleaning, knowledge enrichment, and knowledge deployment to provide a complete life cycle for this process.
- Part IV provides application types and actual applications as well as an outlook on additional trends that will make the need for knowledge graphs even stronger.

This book is intended as a textbook for courses covering knowledge graphs. It introduces the theoretical foundations of the technologies essential for knowledge graphs but also covers practical examples, applications, and tools. We very much invite you to work with this book, and we hope we do not lose you in Part II on semantics and logic. No pain, no gain.

Innsbruck, Austria

Umutcan Serles
Dieter Fensel

References

Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Sci Am 284(5): 34–43

Bonatti PA, Decker S, Polleres A, Presutti V (eds) (2019) Knowledge graphs: new directions for knowledge representation on the semantic web (Dagstuhl seminar 18371). Dagstuhl Reports 8(9)

Chen H, Ji H, Sun L, Wang H, Qian T, Ruan T (eds) (2016) Knowledge graph and semantic computing: semantic, knowledge, and linked big data: First China Conference, CCKS 2016, Beijing, China, September 19–22, 2016, Revised Selected Papers, vol 650. Springer

Croitoru M, Marquis P, Rudolph S, Stapleton G (eds) (2018) Graph structures for knowledge representation and reasoning: 5th International Workshop, GKR 2017, Melbourne, VIC, Australia, August 21, 2017, Revised Selected Papers, LNCS, vol 10775. Springer

d'Amato C, Theobald M (eds) (2018) Reasoning web. In: Learning, uncertainty, streaming, and scalability: 14th International Summer School 2018, Eschsur-Alzette, Luxembourg, September 22–26, 2018, Tutorial Lectures, LNCS, vol 11078. Springer

Ehrig H, Ermel C, Golas U, Hermann F (2015) Graph and model transformation. Monographs in theoretical computer science, Springer

Fensel D, Musen MA (2001) The semantic web: a brain for humankind. IEEE Intell Syst 16(2):24–25

Fensel D, Hendler JA, Lieberman H (eds) (2005) Spinning the Semantic Web: bringing the World Wide Web to its full potential. MIT Press

Fensel D, Simsek U, Angele K, Huaman E, Kärle E, Panasiuk O, Toma I, Umbrich J, Wahler A (2020) Knowledge graphs. Springer

Hogan A, Blomqvist E, Cochez M, d'Amato C, Melo Gd, Gutierrez C, Kirrane S, Gayo JEL, Navigli R, Neumaier S, et al. (2021) Knowledge graphs. ACM Comput Surv 54(4):1–37

Li J, Zhou M, Qi G, Lao N, Ruan T, Du J (eds) (2018) Knowledge graph and semantic computing. Language, knowledge, and intelligence: Second China Conference, CCKS 2017, Chengdu, China, August 26–29, 2017, Revised Selected Papers, vol 784. Springer

Page L, Brin S, Motwani R, Winograd T (1999) The PageRank citation ranking: bringing order to the web. Technical report, Stanford InfoLab

Pan JZ, Calvanese D, Eiter T, Horrocks I, Kifer M, Lin F, Zhao Y (eds) (2017a) Reasoning web: logical foundation of knowledge graph construction and query answering: 12th International Summer School 2016, Aberdeen, UK, September 5–9, 2016, Tutorial Lectures, LNCS, vol 9885. Springer

Pan JZ, Vetere G, Gomez-Perez JM, Wu H (eds) (2017b) Exploiting linked data and knowledge graphs in large organisations. Springer

Van Erp M, Hellmann S, McCrae J, Chiarcos C, Choi K, Gracia J, Hayashi Y, Koide S, Mendes P, Paulheim H, et al. (eds) (2017) Knowledge graphs and language technology. In: Proceedings of the 15th International Semantic Web Conference (ISWC2016): International Workshops: KEKI and NLP & DBpedia, Kobe, Japan, October 17–21, 2016

Acknowledgments

Writing such a comprehensive textbook for knowledge graphs is not easy, and we would like to acknowledge the support of various people who helped us. The core of this book is the lecture slides we prepared for our Knowledge Graph and Semantic Web courses. We would like to thank Anna Fensel, who prepared the Semantic Web course for many years and provided us with a good basis for our course materials. We would also like to thank the PhD students in our team, Kevin Angele, Elwin Huaman, Juliette Opdenplatz, and Dennis Sommer. Their PhD work in the areas of knowledge deployment, knowledge assessment, knowledge cleaning, and knowledge enrichment provided the foundation for the chapters of this book.

We use several real-world examples in the book, especially ones from the tourism domain. Many thanks to Onlim GmbH for giving us early access to the German Tourism Knowledge Graph and for hosting some of the tools we present in this book for the knowledge graph life cycle.

Editing a book this size is a challenging task. We are grateful to our student assistants Muhammed Umar, Shakeel Ahmed, and Elbaraa Elsaadany for their support in putting the book together and Ina O'Murchu for working hard to turn our German/Turkish version of English into the English copy. Finally, we would like to also thank Ralf Gerstner from Springer for being very understanding and supportive during the publication process.

Contents

Part I Knowledge Technology in Context

1 Introduction . . . 3
References . . . 4

2 Artificial Intelligence . . . 5
2.1 General Problem Solver and Limited Rationality . . . 5
2.2 Knowledge Representation . . . 7
2.3 Reasoning with Knowledge . . . 10
2.4 Knowledge Modeling . . . 14
2.5 Ontologies . . . 18
2.6 Summary . . . 23
References . . . 23

3 Information Retrieval and Hypertext . . . 27
3.1 Boolean Model . . . 27
3.2 Vector Space Model . . . 28
3.3 Evaluation of Information Retrieval Systems . . . 31
3.4 PageRank . . . 32
3.5 Applications and Systems . . . 35
3.6 Hypertext . . . 35
3.7 Summary . . . 37
References . . . 37

4 The Internet . . . 39

5 The World Wide Web . . . 45
5.1 Uniform Resource Identifiers (URIs) . . . 46
5.2 REST and HTTP . . . 47
5.3 HTML and XML . . . 49
5.4 Is XML Schema an Ontology Language? . . . 51
References . . . 53

6 Natural Language Processing 55
6.1 An Example: The GATE System 57
References 61

7 Semantic Web: Or AI Revisited 63
References 68

8 Databases 69
References 73

9 Web of Data 75
9.1 Linked Data 75
9.2 Linked Open Data 78
References 83

10 Knowledge Graphs 85
References 87

Part II Knowledge Representation

11 Introduction to Knowledge Representation 91
Reference 92

12 The Five Levels of Representing Knowledge 93
References 96

13 Epistemology 97
13.1 Data Model 98
13.1.1 The Resource Description Framework (RDF) 98
13.1.2 RDF Schema (RDFS) 105
13.1.3 RDF(S) Serialization 107
13.1.4 Tools 113
13.1.5 Summary 114
13.2 Data Retrieval and Manipulation 114
13.2.1 SPARQL 115
13.2.2 SHACL 128
13.2.3 Summary 151
13.3 Reasoning over Data 151
13.3.1 OWL and OWL2 151
13.3.2 Rules 168
13.3.3 Summary 176
13.4 SKOS: A Lightweight Approach for Schema Integration 177
13.4.1 Knowledge Organization Systems 178
13.4.2 Simple Knowledge Organization System (SKOS) 181
13.4.3 Tools and Applications 186
13.4.4 Summary 187
13.5 Summary 188
References 189

14 The Logical Level . . . 191
14.1 Logics . . . 192
14.1.1 Propositional Logic . . . 192
14.1.2 First-Order Logic . . . 203
14.1.3 Description Logic . . . 209
14.1.4 Herbrand Model Semantics . . . 215
14.1.5 Second-Order Logic and Further Logical Variants . . . 220
14.1.6 Reasearch . . . 223
14.1.7 Summary . . . 224
14.2 RDF(S) Semantics . . . 225
14.2.1 RDF(S) Layering . . . 226
14.2.2 RDF(S) Interpretations . . . 227
14.2.3 RDF(S) Entailment . . . 234
14.2.4 Summary . . . 240
14.2.5 Appendix . . . 241
14.3 SPARQL Query Evaluation . . . 246
14.3.1 Algebra . . . 246
14.3.2 SPARQL Algebra . . . 247
14.3.3 Example Abstract Query Evaluation . . . 250
14.3.4 Summary . . . 255
14.4 Summary . . . 255
References . . . 257

15 Analysis of Schema.org at Five Levels of KR . . . 259
15.1 Schema.org at the Conceptual Level . . . 259
15.2 Schema.org at the Epistemological Level . . . 260
15.2.1 Types . . . 261
15.2.2 Properties . . . 262
15.2.3 Inheritance . . . 263
15.2.4 Instantiation Relationship . . . 264
15.2.5 Multityped Entities . . . 264
15.3 Schema.org at the Logical Level . . . 265
15.3.1 Types . . . 265
15.3.2 Properties . . . 265
15.3.3 Inheritance . . . 267
15.3.4 Instantiation . . . 268
15.3.5 Multityped Entities . . . 268
15.4 Schema.org at the Implementation Level . . . 268
15.5 Summary . . . 269
References . . . 270

16 Summary . . . 271

Part III Knowledge Modeling

17 Introduction: The Overall Model . . . 275
References . . . 277

18 Knowledge Creation . . . 279
18.1 Ontologies . . . 279
18.1.1 Domain-Independent Ontologies . . . 280
18.1.2 Domain Ontologies . . . 281
18.1.3 Ontology Engineering and Methodologies . . . 282
18.1.4 Ontology Manifoldness . . . 284
18.1.5 Summary . . . 287
18.2 ABox Creation . . . 287
18.2.1 Domain Specification, a Connector for ABox and TBox . . . 288
18.2.2 Bottom-Up and Top-Down Domain Specification . . . 289
18.2.3 Manual, Semi-Automatic, and Automatic Creation of Knowledge . . . 291
18.2.4 Handling of Dynamic and Active Data . . . 295
18.2.5 Summary . . . 298
References . . . 298

19 Knowledge Hosting . . . 303
19.1 Challenges in Hosting Knowledge Graphs . . . 304
19.2 Knowledge Hosting Paradigms . . . 305
19.2.1 Relational Databases . . . 305
19.2.2 Document Model . . . 312
19.2.3 Graph Databases . . . 315
19.3 RDF Triplestores . . . 316
19.4 Illustration: German Tourism Knowledge Graph in GraphDB . . . 317
19.4.1 Storing an RDF Graph . . . 317
19.4.2 Querying . . . 319
19.4.3 Visualization . . . 319
19.4.4 Reasoning . . . 322
19.5 Summary . . . 324
References . . . 325

20 Knowledge Assessment . . . 327
20.1 Quality Dimensions . . . 329
20.2 Calculating Quality Score . . . 333
20.3 Approaches and Illustration . . . 334
20.4 Summary . . . 336
References . . . 337

21 Knowledge Cleaning 339
21.1 Error Types 340
21.1.1 Wrong Instance Assertions 341
21.1.2 Wrong Equality Assertions 342
21.1.3 Wrong Property Value Assertions 343
21.2 Error Detection and Correction 345
21.2.1 Syntactical Processing 345
21.2.2 Statistical Methods 347
21.2.3 Logical and Knowledge-Based Approaches 348
21.2.4 A Few Frameworks for Error Detection and Correction 351
21.3 Illustration: Cleaning the German Tourism Knowledge Graph 353
21.3.1 Error Detection 354
21.3.2 Error Correction 354
21.4 Summary 355
References 356

22 Knowledge Enrichment 359
22.1 Identification of Additional Data and Knowledge Sources 361
22.2 Data Lifting 361
22.3 TBox Alignment 362
22.4 ABox Integration 368
22.4.1 Entity Resolution 369
22.4.2 Data Fusion 374
22.5 Illustration: Enriching the German Tourism Knowledge Graph 376
22.5.1 Duplicate Detection 376
22.5.2 Data Fusion 377
22.6 Summary 378
References 379

23 Tooling and Knowledge Deployment 383
23.1 Tooling for the Knowledge Graph Life Cycle 384
23.1.1 Knowledge Creation 384
23.1.2 Knowledge Hosting 385
23.1.3 Knowledge Assessment 388
23.1.4 Knowledge Cleaning 389
23.1.5 Knowledge Enrichment 391
23.1.6 Knowledge Deployment 391
23.2 Knowledge Access and Representation Layer 391
23.2.1 Architecture 394
23.2.2 Knowledge Activators 395
23.2.3 Illustration 397
23.3 Summary 405
References 406

24 Summary 409

Part IV Applications

25 Applications 415
25.1 Migration from Search to Query Answering 415
25.2 Virtual Assistants 417
25.3 Enterprise Knowledge Graphs 422
25.3.1 Data and Knowledge Integration Inside an Enterprise 422
25.3.2 Data and Knowledge Exchange in Enterprise Networks 424
25.4 Cyber-Physical Systems and Explainable AI 430
25.5 Summary 432
References 433

Part I
Knowledge Technology in Context

Chapter 1
Introduction



The developments that led to knowledge graphs go all the way back to the 1940s and earlier. We will cover these related fields briefly for several scientific areas (see also Harth 2019, Gutierrez and Sequeda 2020, Pavlo 2020, Sanderson and Croft 2012):

- *Genuine areas*: artificial intelligence (AI), particularly symbolic AI as well as Semantic Web and Linked Open Data (LOD), i.e., Web of Data
- *Heavily related areas*: information retrieval (IR), hypertext systems, natural language processing (NLP), the Internet, the World Wide Web (WWW), and databases (DB)

Figure 1.1 provides a roadmap of these various fields, which we will discuss in the following. The purpose is to provide an understanding of technologies essential for building knowledge graphs. They did not fall from the sky but are instead a result of a large collection of related fields.

Disclaimer We do not try to provide a complete survey on all these fields. This would be far beyond our capabilities and purpose. We instead focus on aspects of these fields that were essential, enabling technologies for knowledge graphs. Therefore, our view on these other fields is biased, and we would like to beg the pardon of our colleagues working in these areas.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_1

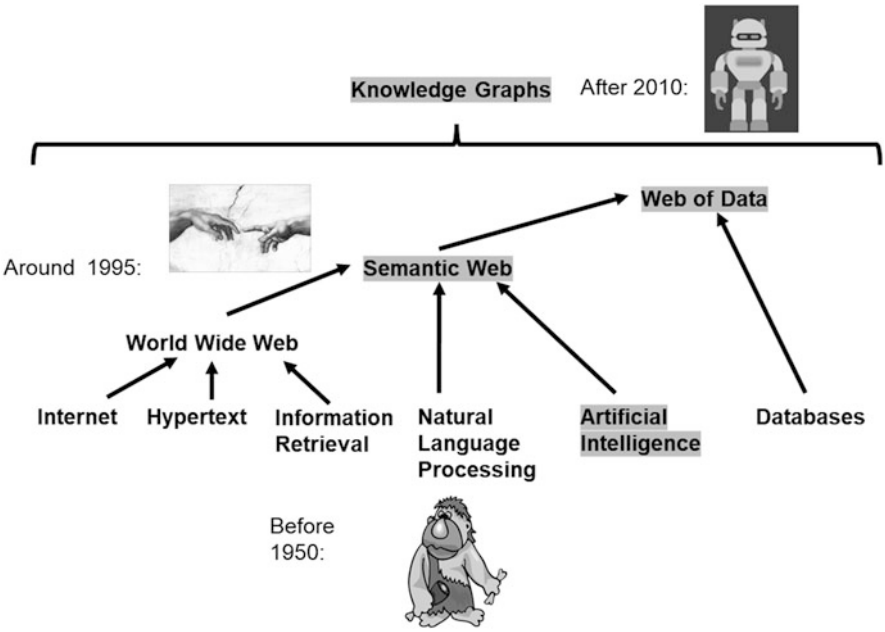


Fig. 1.1 The tree of evolution of knowledge graph technology (Robot graphic by Vectorportal.com licensed with CC-BY 4.0 https://creativecommons.org/licenses/by/4.0/)

References

Gutierrez C, Sequeda JF (2020) Knowledge graphs: A tutorial on the history of knowledge graph's main ideas. In: Proceedings of the 29th ACM international conference on information & knowledge management (CIKM '20), Virtual Event Ireland, October 19–23, pp 3509–3510

Harth A (2019) Introduction to linked data, Part I. Preview Edition

Pavlo A (2020) 01 - History of databases (CMU databases/Spring 2020). https://www.youtube.com/watch?v=SdW5RKUboKc

Sanderson M, Croft WB (2012) The history of information retrieval research. In: Proceedings of the IEEE 100 (Special Centennial Issue):1444–1451. doi: https://doi.org/10.1109/JPROC.2012.2189916

Chapter 2
Artificial Intelligence



We first describe the concept of general problem solvers and infer from their limits the need for limited rationality and heuristic search. Then we introduce the concept of "knowledge is power," which led to new research fields such as knowledge representation, reasoning engines following various paradigms, expert systems, as well as methods and techniques to build them based on engineering principles.

2.1 General Problem Solver and Limited Rationality

Gottfried Leibniz, a mathematician and philosopher who lived in the seventeenth century, believed that every rational reasoning of humans could be reduced to calculations. He theorized a calculus ratiocinator,[1] a machine that can solve any problem written in a "universal language." There are different views on what he meant by this:

- One view sees it as an inference engine,
- The other view as a calculating machine, and
- For him, inference and calculation were identical.

Leibniz later prototyped the Stepped Reckoner, a machine that solves differential equations, which is often attributed to the calculus ratiocinator idea.

A significant event that "started" artificial intelligence (AI) is identified as the Dartmouth Workshop (McCarthy et al. 2006). It was an 8-week workshop organized and participated by scientists and industrial practitioners from various fields, such as computer science, cognitive science, mathematics, and natural sciences, which was held in 1956 at Dartmouth College in New Hampshire, USA. The main objective was to discuss the ideas around "thinking computers." At this workshop, the idea to

[1] https://en.wikipedia.org/wiki/Calculus_ratiocinator

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_2

solve all problems that require intelligence with a *General Problem Solver* was (re)born (Ernst and Newell 1969; Newell and Shaw 1959; Newell and Simon 1972; Newell et al. 1959).

A General Problem Solver is a system that aims to solve all problems with deduction as long as these problems can be represented formally (e.g., with formal logic). Given an initial and a goal state, it searches the path from the initial state to the goal.[2] It was developed in 1959 following the Dartmouth Workshop by A. Newell, J. C. Shaw, and Herbert A. Simon. Actually, it worked well for many "toy" problems that also can be formalized straightforwardly. The general algorithm is given a state, and an operator is applied that takes you closer to your goal state. Even a "simple" problem like playing chess gets lost in a combinatorial explosion. Shannon conservatively estimated the lower bound number of possible games as 10^{120} (Shannon 1950). For a complete and correct search, you need to evaluate all of them.[3]

Newell and Simon realized that the combinatorial explosion would be a problem. All relevant and interesting problems come with search spaces that are incredibly large, if not infinite. A blind brute-first search of general problem solvers does not scale at all. They introduced the concept of *limited rationality* and *heuristic search*.

- *Limited rationality* is giving up on the goal of finding an optimal solution by considering the costs of finding it (Simon 1957; Newell and Simon 1972).
- *Heuristic search* implements this in a way that you limit your search space and use *heuristics* to find acceptable semi-optimal solutions (local instead of global optima).[4]

Hill climbing is an example of a heuristic search algorithm.[5] Given a *solution state space of a problem and a heuristic function*, the hill climbing algorithm traverses from its current state to a neighboring state with the aim of reaching the goal state. The complete state space is not necessarily known in advance. The *heuristic function* calculates the "heuristic value" of a state. A state is visited only if it appears to bring us closer to the goal state (decided with the help of having the highest heuristic value from all neighborhood nodes). A typical example is the traveling salesman problem, which is about visiting a number of interconnected places with a minimal total path length. The complete algorithm would compute all possible paths and select the shortest one. However, with a complete search, we run into a combinatorial explosion. With hill climbing, we cannot guarantee that we will find the best solution but we will find a local optimum much faster. We always

[2] The search can start from the goal state backward, from the start state forward or from an intermediate state.

[3] https://electronics.howstuffworks.com/chess.htm#pt1 and https://medium.com/hackernoon/machines-that-play-building-chess-machines-7feb634fad98

[4] https://en.wikipedia.org/wiki/Heuristic_(computer_science)

[5] https://en.wikipedia.org/wiki/Hill_climbing

would move to the next place in the neighborhood that is close to our current state. Obviously, we cannot guarantee that we found the shortest path in the end.

Up to now, we discussed how difficult it is to find the global optimum. Actually, the problem may start even earlier: It may be difficult to decide what the global optimum is. And even worse, it may change over time. A nice example is provided by climbing to the top of a mountain.[6] The following problems may appear:

- It is hard to correctly identify a mountain's peak, especially with snow cover on the ground and bad weather conditions. Many mountain climbers who are assumed to be on the highest point of a mountain may have missed this point by some meters. *So finding the global optima is extremely hard.*
- Even worse, the highest point of a mountain changes over time. So you may have been at the highest point in the past but not in the present. *Global optima may change quickly over time.*

General problem solvers searching the entire search space do not scale. The concept of the incomplete heuristic search was a response to it. However, they are not complete and, therefore, may not find an optimal solution for the general case. For example, the algorithm may terminate at a local optimum. We can implement backtracking, generate more (or all!) states, and so on with the hope of reaching a global optimum. However, then we are just back to a complete search. The question is whether the time and effort are worth it or whether a local optimum may be good enough. Recall limited rationality and the idea of including the costs for finding a solution into consideration.

Knowledge Is Power In essence, you need *knowledge* to guide your search properly. Feigenbaum formulated this as the knowledge principle (Lenat and Feigenbaum 1987):

> A system exhibits intelligent understanding and action at a high level of competence primarily because of the specific knowledge that it contains about its domain of endeavor.

The question is how to represent this knowledge and how to work with this knowledge to solve problems. The proof of the pudding is in the eating: expert systems were built to prove his point. The three major pillars for implementing this slogan are knowledge representation, reasoning with knowledge, and knowledge modeling.

2.2 Knowledge Representation

How is knowledge represented? Soon a number of alternative paradigms arose. Since this was a parallel development, many systems and representation formalisms had features of several paradigms. Still, we can distinguish the following main

[6] Frankfurter Allgemeine Zeitung, 13.10.2021, Nr. 238, S. 7: *Das ist doch der Gipfel*.

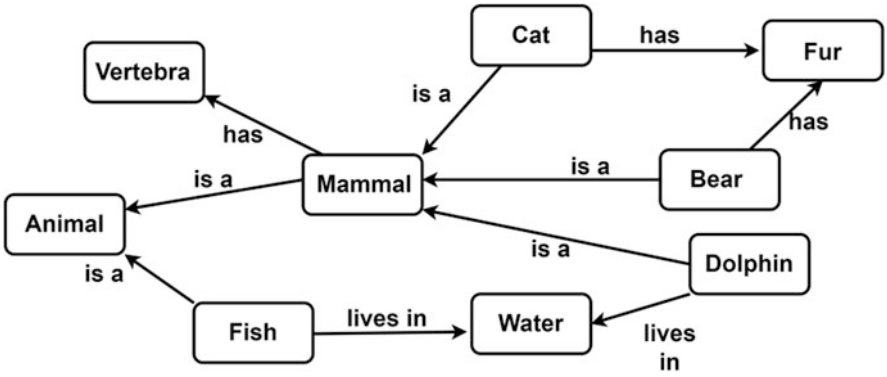


Fig. 2.1 The term knowledge graph was first used to describe a semantic net whose edge labels are restricted to a certain vocabulary (James 1992). In the figure, you see an example of a semantic network (Adapted from https://en.wikipedia.org/wiki/Semantic_network)

streams: *semantic networks* that led to description-logic-based representation formalisms. The network character also reflected the later structure of the World Wide Web. *Frames* took a more local view of an object with defined properties and property values and became a predecessor of object-oriented programming. A less dominant approach was *prototypes,* where knowledge is described by characterizing properties of sets of entities.

"A *semantic network* is a structure for representing knowledge as a pattern of interconnected nodes and arcs" (Sowa 1991). It was first used in 1960 for natural language processing (NLP) tasks, particularly to help the machine translation of different languages. Over the years, many different versions of it have been developed. Common characteristics are listed in Sowa (1991) as follows (see also Fig. 2.1):

- Nodes in the net represent concepts.
- Arcs in the net represent the relationships between concept nodes.
- Concepts are hierarchically organized.
- And relationships are inherited across the concept hierarchy.

Frames (Minsky 1975) are also networks of nodes and relationships. Knowledge is stored in a structure called a "frame." These frames can be organized in a hierarchy and linked via slots. Constraints and conditions can be defined on a slot, as well as actions can be defined based on certain triggers. They also allow the representation of "stereotypical knowledge." The top levels of a frame are always true for a given situation. Lower levels have multiple slots whose values are filled with specific instances of data. That is, lower-level frames may override the default value inherited from a higher-level frame (see Fig. 2.2).

Semantic nets and frame-based systems both represent nodes and their relationships; however, they use different representation paradigms (Grimm et al. 2007). The semantic net paradigm represents arbitrary nodes and arbitrary binary

Slot	Value	Type
BOY	_	(This Frame)
ISA	Person	(parent frame)
SEX	Male	(instance value)
AGE	Under 12 yrs.	(procedural attachment - sets constraint)
HOME	A Place	(frame)
NUM_LEGS	Default = 2	(default, inherited from Person frame)

Instance

Slot	Value	Type
ALEX	_	(This Frame)
NAME	Alex	(key value)
ISA	Boy	(parent frame)
SEX	Male	(inheritance value)
AGE	IF-NEEDED: Subtract(current,BIRTHDATE);	(procedural attachment)
HOME	100 Main St.	(instance value)
BIRTHDATE	8/4/2000	(instance value)
FAVORITE_FOOD	Spaghetti	(instance value)
CLIMBS	Trees	(instance value)
BODY_TYPE	Wiry	(instance value)
NUM_LEGS	1	(exception)

Fig. 2.2 An example of frame-based class, slot, and instance definitions [Example adapted from https://en.wikipedia.org/wiki/Frame_(artificial_intelligence)]

relationships between them. The relationships are represented as edges between two nodes. The frame-based paradigm sees nodes as frames and relationships as slots defined on frames filled by other frames (akin to the object-oriented paradigm). As a result, frames provide more structure to nodes. Frames also have a more natural way to represent exceptions in the inheritance of relationships, i.e., the value of an inherited slot can be easily overridden by the inheritor. In semantic nets, consuming applications need to decide on a strategy to solve such inheritance conflicts.

Prototypes (Cochez et al. 2016) are a variation of frames. Usually, they have no notion of classes; instead, there is a prototype frame. The instances are defined based on the prototype frame and inherit properties from a base prototype. It can add/change/remove property values. A variation of fuzzy logic or a similarity function can be applied to retrieve similar instances.

2.3 Reasoning with Knowledge

Besides representing knowledge, one would like a computer to process this knowledge drawing new conclusions from it. We identify three major paradigms for this kind of engine:

- *Description logics* that base their semantics on standard first-order logic (FOL)
- *Rule systems* that base their semantics on variations of minimal model semantics of Horn logic (logic programming)
- *Production rule systems* that follow a more procedural style, often recalling the assembly's go-to style

A **description logic (DL)** knowledge base consists of two parts: TBox and ABox. A *terminological box* (*TBox*) contains axioms about concept hierarchies and roles (relationships). An *assertional box* (*ABox*) contains ground facts, i.e., instances of concepts and relationships. There is a *strict* separation between these two parts, mostly in order to provide scalable reasoners. The main conceptual building blocks are concepts (unary predicates), roles (binary predicates), and individuals (constants). Like first-order logic, a DL uses the open-world assumption (OWA) and does *not* have a unique name assumption (UNA).

Open-World Assumption (OWA) The nonexistence of a statement does not mean it is false. The implication of OWA is that if a statement is false, it has to be explicitly stated. Imagine you are in a train station and there are only the following entries in the timetable:

Innsbruck – Munich 13:00
Innsbruck – Kufstein 13:10

Is there a train connection between Innsbruck and Munich at 15:00? Intuitively, your answer would be "no." Under OWA, the answer cannot be "no" as we do not know if the given connection does not exist. Train station schedules work under the ***closed-world assumption***, which makes our lives easier. If you do not see a train connection in the timetable, it does not exist. Because of this, FOL, and therefore DL, cannot express the transitive closure of a relation (see Part II for more details).

Unique Name Assumption (UNA) Things that have different names (identifiers) are different. This is the implication of having UNA. Take the following axioms: a person can only be married to another unique person.[7] Now, consider we have the following ground facts: *married(john,sally)* and *married(john,sam)*. A knowledge base with UNA would be inconsistent as *sally* and *sam* are different individuals. A DL system not having the UNA would infer that *sally* and *sam* refer to the same individuals. Nothing stops different names from referring to the same individual

[7] Obviously, this constraint does not hold in all cultures; also, there is the problem that people marry again in time, and finally, the constraint we were taught at university that each spouse must have a different gender is gone, too.

TBox	
$Bachelor \equiv \neg\exists married.\top \sqcap Man$	Bachelors are unmarried men and vice versa
$married \equiv married^{-1}$	Being married to someone is reflexive
$\exists_{\geq 2}\ married.Person \sqsubseteq \bot$	One can be married to at most one person
ABox	
Man(umut)	Umut is a Man
Woman(miriam)	Miriam is a Woman
married(umut, miriam)	Umut is married to Miriam

Fig. 2.3 An example of a specification in description logic [Adapted from Rodriguez (2019)]

unless it is explicitly stated that they are different. An important implication would be on the resolution algorithm for proving a conclusion from premises. A crucial step in resolution is unification (more on this in Sect. 14.1). Only expressions with the same term symbols can be unified. Normally, this is a simple linear syntactic term comparison. Without a unique name assumption, we need to "reason" whether f and g are the same terms, even if they look syntactically different.

Usually, DL systems focus on a decidable subset of first-order logic, but reasoning can still be very expensive. The most basic DL language that allows negation, conjunction, and disjunction is an attributive language with complements (ALC). For example, the satisfiability problem of a TBox for ALC has *exponential time complexity* w.r.t size of the TBox. It becomes a *polynomial space complete* if no cycles in the TBox are allowed (Martinez et al. 2016; Ortiz 2010). A small DL knowledge base is shown in Fig. 2.3.

The description logic community is a vibrant community with many developed systems (e.g., Hermit, Pellet, RACER, etc.) and applications in areas such as software engineering, configuration tasks, medical informatics, Web-based information systems, natural language processing, etc.; see Baader et al. (2003). For example, GALEN (Rector and Nowlan 1994) was a large European project to provide a common reference model for medical terminology. The model contains modules for diseases, causes, human anatomy, genes, etc. The model is formalized with DL and aims to support applications like decision support systems and the natural language processing of medical text. Finally, the Web ontology language (OWL) development was a big boost for it.

Prolog stands for *PROgramming in LOGic*, which is a logic programming language (framework) invented in 1972.[8] It is based on Horn clauses written in implication form. The program tries to answer a query (the goal) via logical deduction (see also Coppin (2004) for a brief introduction to Prolog and Horn clauses).

[8] https://en.wikipedia.org/wiki/Prolog

A Horn clause in logical programming is a formula that looks like a rule (IF... Then ...). It is a disjunction of literals with at most one positive literal.

$\neg p \vee \neg q \vee \ldots \vee \neg t \vee u$, which can be written in implication form as $u \leftarrow p \wedge q \wedge \ldots \wedge t$

It uses minimal model semantics and its closed-world assumption by negation as failure, i.e., nonderivable facts are false. A program in Prolog is a database of facts and rules:

- A rule consists of a head and a body:
  - The body of a rule is a sequence of compound terms separated by a comma (,) or a semicolon (;).
  - The body implies the head.
  - The rules are written as implications: head: - body.
  - For example, *sibling(X, Y): - parent_child(Z, X), parent_child(Z, Y).*
- Facts are rules without a body:
  - For example, *father_child(mike, tom).*

A program is run by giving a goal (a query). The program tries to answer the query via logical deduction. Prolog uses a proof process called resolution in order to answer queries based on facts and rules. It is actually a refutation process. We first include the negation of the goal to the premises and then prove that the set of formulas is unsatisfiable. Resolution works on the principle that a literal and its negation can be "unified" under certain conditions:

- Can we make these two literals identical by substituting their variables with other terms?
- If yes, then we can resolve them.
- If we can reach the empty clause, then the query is answered as true.

The example in Fig. 2.4 shows the resolution algorithm in action (Williams 2005). First, the query *ancestor(P, juliette)* is resolved with the rule (1). This is done by the substitution $\theta_1 = \{P/X, Y/ juliette\}$ to obtain the subgoal *parent(X, juliette)*, where *X is mapped* to *P* and *juliette is mapped* to *Y. Then r*esolve subgoal *parent(X, juliette)* with the fact (4). This is done with the substitution $\theta_2 = \{X/ george\}$ to yield the empty clause. The answer is $P = george\}$, i.e., *george* is an ancestor of *juliette*.

Prolog provides the *not* operator that can be used in the body of the rule to indicate negation. The negation in Prolog works with the *negation as failure (NaF)* principle. The literal *not(P(x))* holds true if *P(X)* cannot be derived; otherwise, it is false (Basic and Snajder 2019). Negation as failure (NaF) implements the closed-word assumption.

```
(1) ancestor(X,Y) :- parent(X,Y).
(2) ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
(3) parent(morty,jerry).
(4) parent(george, juliette).
(5) parent(jerry,sue).
(6) parent(jerry,alan).
(7) parent(frank,george).
(8) parent(sue,toby).
(9) parent(sue,juliette).

(Query) :- ancestor(P,juliette).
```

Fig. 2.4 An example Prolog program and a query

```
(1) human(X) :- speaks(X), not(has_feathers(X)).
(2) speaks(plato).
(3) speaks(polynesia).
(4) has_feathers(polynesia).

? – human(plato) .
```

Fig. 2.5 An example of negation as failure

The example in Fig. 2.5 explains NaF (adapted from Basic and Snajder (2019)). We cannot derive *has_feathers(plato))*; therefore, *not(has_feathers(plato))* would be true. As a consequence, from the fact (2) and rule (1), we can infer that *human(plato)* holds true.

Production Rule Systems Many expert systems were implemented as *production rule systems*. A production rule system consists of a rule base in the form of a set of condition-action rules, a working memory that holds the state of the system, and a rule engine that orchestrates the triggering of the rules.

MYCIN[9] was an expert system implemented as a production rule system in the 1970s. It contained about 350 production rules that encoded clinical diagnosis

[9] https://en.wikipedia.org/wiki/Mycin

PREMISE: ($AND (SAME CNTXT GRAM GRAMNEG)
(SAME CNTXT MORPH ROD)
(SAME CNTXT AIR ANAEROBIC))
ACTION: (CONCLUDE CNTXT IDENTITY BACTEROIDES TALLY .6)

If 1) The gram stain of the organism is gramneg, and
2)The morphology of the organism is rod, and
3) The aerobicity of the organism is anaerobic

Then: There is a suggestive evidence (0.6) that the identity of the organism is bacteroides

Fig. 2.6 A MYCIN rule (Van Melle 1978)

criteria from infectious disease experts. Its main purpose was to help physicians with their decisions on bacterial disease diagnosis. The rule system was written in a list-processing language (LISP) (van Melle 1978).[10] A MYCIN rule and its English translation are shown below in Fig. 2.6.

Based on the data in the working memory, the rules whose premises are fulfilled can be triggered. The data are represented in a triple format and can be acquired:

- As static factual data
- As dynamic patient data (collected via consultation), and
- As example patient data

MYCIN has a backward-chaining rule engine. At any given moment, MYCIN is trying to prove a goal (e.g., identify a bacteria). It brings the rules whose conclusion contains the goal and creates subgoals based on the premises. Each conclusion updates the working memory, which would lead to the firing of other rules, forming a rule chain. MYCIN also contains rules about rules, i.e., metarules that enable the prioritization of rules and conflict resolution.[11]

2.4 Knowledge Modeling

"In artificial intelligence, an expert system is a computer system emulating the decision-making ability of a human expert" (Jackson 1998). Early knowledge-based systems like expert systems were developed not with a methodology but rather with rapid prototyping approaches. The main problem with first-generation expert systems was the lack of reusability of knowledge. This made them very hard

[10] See also Coppin (2004).

[11] See Buchanan and Shortliffe (1984) for a book about MYCIN that is compiled based on various related publications.

to develop and, in turn, quite costly. It is comparable to programming from scratch. In response to this, knowledge engineering was developed as a modeling task to structure the building and maintenance process of experts or knowledge-based systems. The need for a methodology for getting the knowledge from the expert's head and formalizing it in a computer system (i.e., knowledge acquisition) has led to the development of a set of methodologies.

A prominent approach in this area of knowledge engineering methodologies is *KADS* (Wielinga et al. 1992) and its successor, *CommonKADS* (Schreiber et al. 1994). KADS and CommonKADS see knowledge acquisition as a modeling task and not as a task of transferring expert knowledge as a flat list of rules. A major principle is to first concentrate on conceptual modeling, then in a later step on the implementation details. Knowledge and how it can be used within a knowledge-based system are modeled from different aspects. CommonKADS provides knowledge-based system developers with a set of model templates that can be configured and filled during the project.

The original situation was very comparable to the situation in computer engineering in the 1970s. Small programs were written quickly and successfully. Larger programs failed horribly in terms of the time of delivery, development costs, and functionality mismatched requirements, and the error rate of programs was much too high. It was especially NASA that was to launch initiatives to improve this situation and help found the field of software engineering.

Just take the Mars Climate Orbiter[12] as an example. It was a robotic space probe launched by NASA. However, communication with the spacecraft was permanently lost as it went into orbital insertion. The spacecraft encountered Mars on a trajectory that brought it too close to the planet, and it was either destroyed in the atmosphere or escaped the planet's vicinity and entered an orbit around the sun. An investigation attributed the failure to a measurement mismatch: metric units by NASA and US Customary (imperial or 'English') units by spacecraft builder Lockheed Martin.

Central means of software engineering are various models that guide the software development process. KADS/CommonKADS does this for a specific type of system: knowledge-based systems. The core models are shown in Fig. 2.7 and are defined by Schreiber et al. (1994) as follows:

- The *organization model* supports the analysis of an organization. The goal is to identify problems, opportunities, and potential impacts of the KBS (knowledge-based system) development.
- The *task model* describes tasks that are performed or will be performed in the organization.
- The *agent model* describes the capabilities, norms, preferences, and permissions of agents. An agent is the executor of tasks.
- The *knowledge model* gives an implementation-independent description of the knowledge involved in a task.

[12] https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

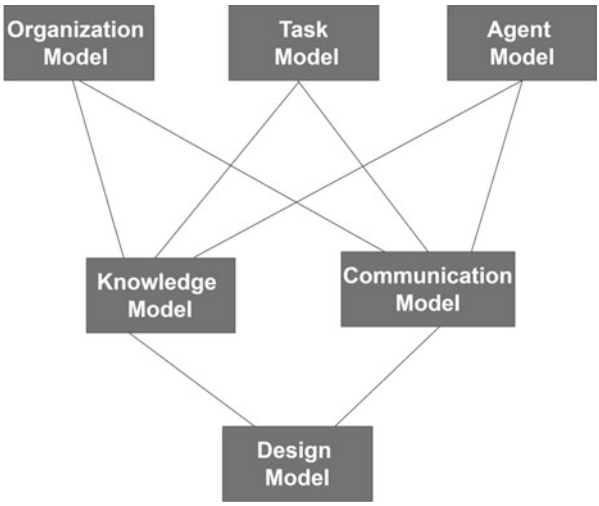


Fig. 2.7 CommonKADS models, adapted from Fig. 1 in Schreiber et al. (1994)

- The *communication model* describes the communicative transactions between agents.
- The *design model* describes the structure of the system that needs to be constructed.

We will go a bit deeper into the knowledge model as it is the most significant one in the context of knowledge graphs. The knowledge model distinguishes between domain, inference, and task knowledge.[13]

Domain knowledge represents the relevant domain knowledge and information, mostly static by nature. See the example for car components in Fig. 2.8.

Inference knowledge models inference actions that are basic reasoning steps that can be made with the domain knowledge and are applied by tasks. From a functional point of view, it describes the lowest level of decomposition. Its basic information-processing units are inference functions that enable reasoning and the transfer function that facilitates communication with other agents. See the inference action *cover* that tries to derive a cause for a malfunction in Fig. 2.9.

Task knowledge is goal oriented and provides a higher level of functional decomposition. It describes goals such as:

- Assessing a loan application to reduce the risk
- Finding the reason for a malfunction of a device and restoring it
- Designing an elevator for a skyscraper

It is typically described in a hierarchical fashion based on reusable design patterns. It is finally mapped on reusable problem-solving methods by refining the latter in task-specific terms and adaptations (see Fig. 2.10).

[13] See also https://commonkads.org/knowledge-model-basics/

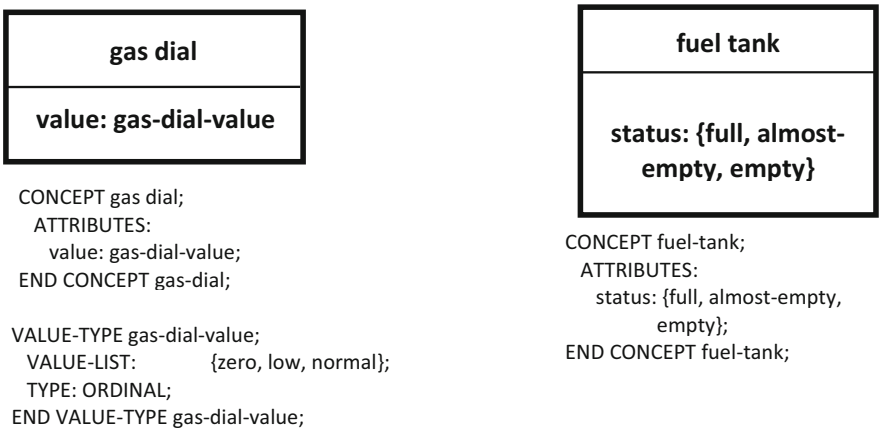


Fig. 2.8 Domain knowledge for car components (Adapted from https://commonkads.org/knowledge-model-basics/)

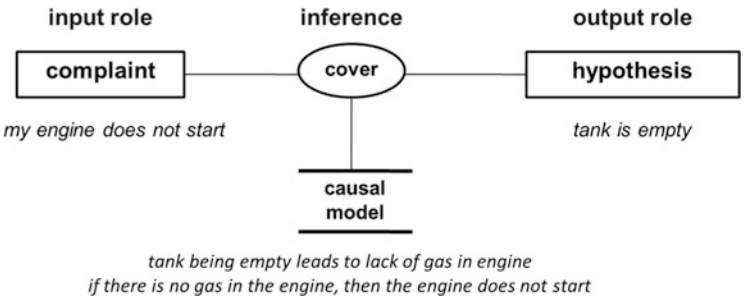


Fig. 2.9 An inference action (Adapted from https://commonkads.org/knowledge-model-basics/)

KADS/CommonKADS provides a methodological framework for developing knowledge-based systems. It approaches KBS development as a modeling task, avoiding the drawbacks of seeing such development as mining knowledge from experts' heads. Its focus is on reusability. It considers many aspects, such as domain, task, organization, and agent models. Here, we focused on domain and task knowledge, but for further details about the methodology, see Schreiber et al. (1994). The core principles of these methodologies are:

- Reusable domain knowledge: *ontologies*
- Reusable task knowledge: *problem-solving methods*

This differentiation generalizes the distinction between *data and programs* but is also aiming at keeping pieces of both aspects reusable. We will focus here on the aspect of data.

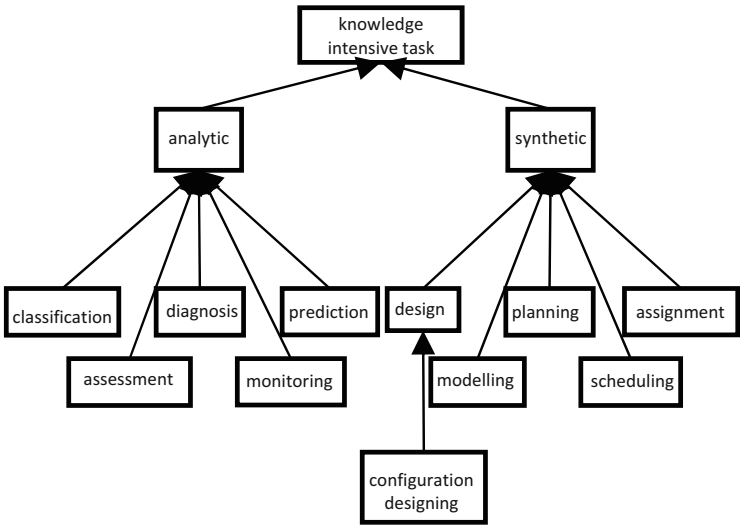


Fig. 2.10 The task model, adapted from Schreiber et al. (1994)

2.5 Ontologies

Ontology[14] is the branch of philosophy that studies concepts such as existence, being, becoming, and reality. It includes the questions of how entities are grouped into basic categories and which of these entities exist on the most fundamental level. Ontology is sometimes referred to as the *science of being*. Aristotle, 384–322 BC, is often referred to as the founder of this area of science. Actually, the goal is to build a model of the world.

Unfortunately, nature is not very keen on being classified into fixed schemas. On the contrary, it is full of transitions, overlaps, exceptions, and changes. These are intrinsic features of self-evolving systems that necessarily develop through gradual self-modification rather than falling from the sky with a given blueprint. Let us take an example from Wikipedia. Mammals[15] (from the Latin word *mamma*) are characterized by the presence of mammary glands, which in females produce milk for feeding (nursing) their young. They reproduce with viviparity, i.e., the embryo is developed inside the body of the parent.[16] These characteristics distinguish them from Sauropsida (reptiles and birds), from which they diverged over 300 million years ago. They are oviparous, a reproductive mode in which females lay developing eggs that complete their development and hatch externally from the mother.[17] And

[14] https://en.wikipedia.org/wiki/Ontology
[15] https://en.wikipedia.org/wiki/Mammal
[16] https://en.wikipedia.org/wiki/Viviparity
[17] https://en.wikipedia.org/wiki/Oviparity

then biologists traveled upside down. They found a strange animal, which they called a *duck-billed platypus*, in Australia. This animal brings the offspring to life as eggs but later starts to feed them with milk.[18] The clear separation was gone.

Fish live in water, and then we meet flying fish, fish living on land, moving forward and backward between land and water, etc. Vice versa, crocodiles live most of their time in the water but are not fish. Birds can fly, and then you run into a *Struthio camelus*. Classifying nature is a Sisyphus task.

A second fundamental objection against an objective model of the world is that we also see it from a specific perspective. We are part of it, and our perception has to focus on a small aspect of it.

Plato (428/427 or 424/423–348/347 BC) raised the question of whether we can really see the real world in its essence or only its shadow in *its appearance*. A concrete example is the movement of the stars on the firmament. What we see are stars rotating around the earth. Meanwhile, we know that more realistic models are that this movement only reflects the rotation of the earth.

Kant (1724–1804 AC), a German philosopher, came in a principled way back to this question: Can we really see the real world in its essence or only its shadow in our perceptions? Kant argued that the sum of all objects, the empirical world, is a complex of appearances whose existence and connection occur only in our representations:

> And we indeed, rightly considering objects of sense as mere appearances, confess thereby that they are based upon a *thing in itself*, though we know not this thing as it is in itself, but only know its appearances, viz., the way in which our senses are affected by this unknown something. (Kant 1783 paragraph 32)

In general, perception is selection and construction. Only tiny aspects of the world can be perceived, depending on the perceptive system and the attendance of the observer. In addition, the receiver builds a model to organize and interpret otherwise random signals into a "meaningful" picture that helps him find his pathway to reality (see Fig. 2.11). Kant goes even so far as to postulate space and time as categories of perception and not of the world as such.

If we cannot access – according to Kant – the *thing itself,* why should we assume there exists something at all beyond our cognition? What is it that is in our perception?

Postmodernism takes this indeed a step further by assuming that there is no reality beyond our perception, and there is no sense in talking about it in scientific terms since we have no access to it. We recently saw in the United States where this leads in political debates. There are no facts, and all news I do not like is fake news! You just need to repeat a lie as often as you can till people believe it and it becomes true.

[18] https://en.wikipedia.org/wiki/Platypus

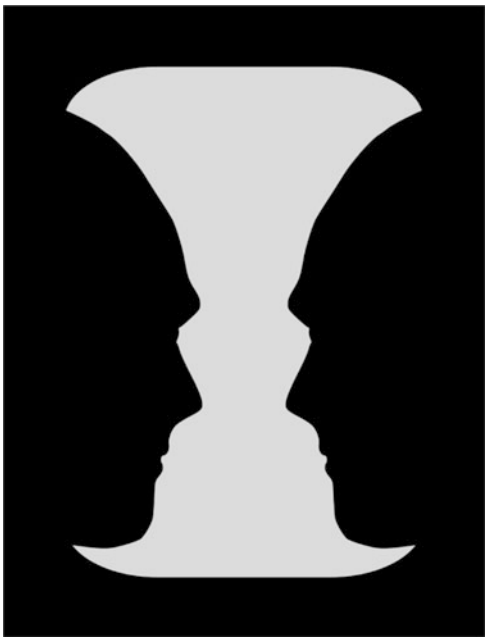


Fig. 2.11 Different perspectives in viewing an object (Image by Nevit Dilmen, licensed with CC-BY-SA 4.0 https://commons.wikimedia.org/wiki/File:Face_or_vase_ata_01.svg)

How did Kant fix this obvious problem of the correspondence between perception and reality? He (reasonably) assumed that there is a world as such, but how can he explain that what is in our perception meaningfully corresponds with some aspect of the world? He assumed a common ground between both. God created our perceptional system and the world in a way that both fit each other. Meanwhile, we would no longer accept this as a scientific answer but as a way to hide away the issue.

However, we also have an answer to this question: "It is the purpose, stupid." Perception and cognition are a means to ensure survival in evolution. They must be able to produce fast and reliable models of the world (but not the world as such) to help feed and for survival.[19] It is actually always the purpose that defines the specific way of modeling the real world. For example, in the river and coastal navigation, your ontology shows a flat world; see Fig. 2.12.

And when changing to offshore sailing (or flying), suddenly everything changes. The earth is suddenly no longer flat, and the shortest path is no longer a straight line (see Fig. 2.13).[20] The difficulty of producing a two-dimensional projection of this three-dimensional sphere became obvious with two-dimensional projection that can either be angle preserving or area size equivalent but never both (see Fig. 2.14). Most of these models are from the first view to support navigation, but they work for certain purposes only.

[19] https://en.wikipedia.org/wiki/Charles_Darwin

[20] https://en.wikipedia.org/wiki/Great-circle_distance

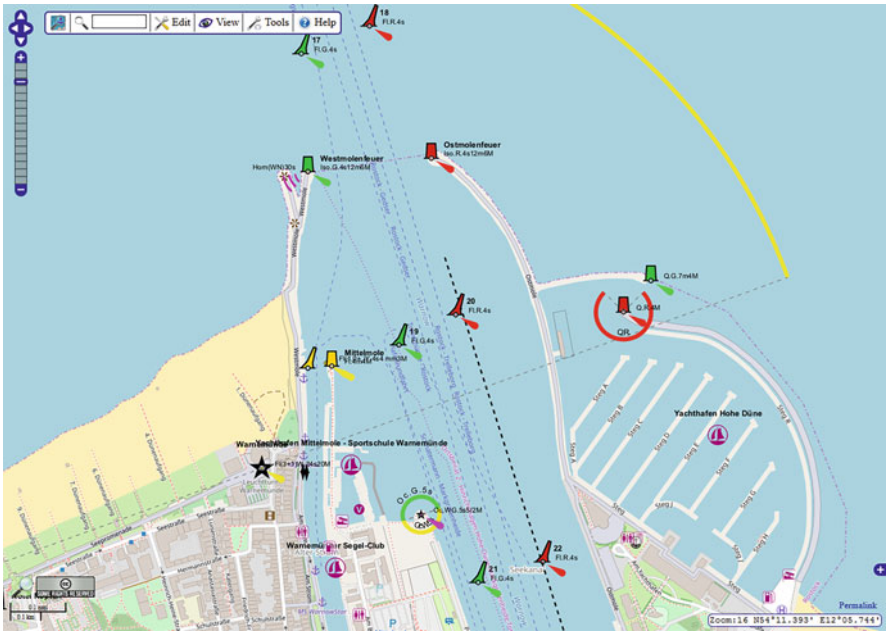


Fig. 2.12 Different perspectives in viewing an object. An ontology for river and coastal navigation models a flat world (Image by OpenSeaMap published by Markus Bärlocher, CC-BY-SA 2.0 https://commons.wikimedia.org/wiki/File:OpenSeaMap-Warnem%C3%BCnde.png)

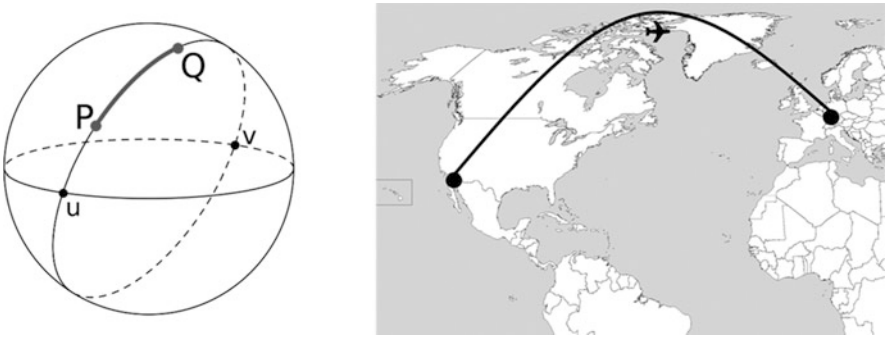


Fig. 2.13 Only in a flat world is the straightway the shortest one. A straight route on earth appears as an arc in a two-dimensional projection like a map (The image on the left by CheCheDaWaff, distributed under CC BY-SA 4.0 https://commons.wikimedia.org/wiki/File:Illustration_of_great-circle_distance.svg. The image on the right derived from the work of MixoMiso27 and distributed under CC-BY-SA 3.0 https://commons.wikimedia.org/wiki/File:A_large_blank_world_map_with_oceans_marked_in_blue.PNG)



Fig. 2.14 The Mercator projection with the wrong size continents (Image by Strebe, distributed under CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:Mercator_projection_Square.JPG)

Let us use the Newton's laws of gravity as a final example. These laws perfectly describe with a small set of differential equations the movement of all bodies in our solar systems. They are so accurate that the existence and location of Neptune could be predicted by the orbital disturbances Uranus has through it. In the same way, researchers predicted a planet called Vulcan to explain the perihelion movement of Mercury. In difference to Neptune, this planet could never be found. In fact, it can only be explained by a more complex theory based on field equations as a mathematical formulation of the space-time continuum. This general relativity theory of Einstein is, for example, also essential to making GPS work.

Summary of Ontologies An ontology always reflects the point of view of an observer. The more this is made explicit by modeling the underlying assumptions, the better it can be aligned with different ontologies. Ontologies can only be partially reused for cases with similar points of view (purpose) in related domains. The idea of *one* right ontology is misleading. It creates pointless arguments and does hamper reuse and alignment.

In computer science, ontologies are used to model things in a certain part of the world from a certain point of view. Ontologies for computer science were defined in the first half of the 1990s (Gruber 1995; Studer et al. 1998) as *a formal, explicit specification of a shared conceptualization*:

- *Conceptualization*: an abstract model of some part of the world
- *Formal*: machine-understandable (e.g., formalized with logic)
- *Explicit*: concepts and constraints are explicitly defined, and
- *Shared*: captures the knowledge on which a group of people have a consensus

2.6 Summary

Still, with all these developments, knowledge acquisition remained a hard and expensive task. The costs related to it often significantly outnumber the gains using an expert system in production. The term *knowledge acquisition bottleneck* was coined. The reasons for the knowledge acquisition bottleneck were:

- Too high costs to acquire knowledge
- The low quality of the acquired knowledge
- Brittleness of the acquired knowledge, and
- The velocity of the acquired knowledge

As a result, this led to the so-called *AI winter* in the 1990s.

References

Baader F, Calvanese D, McGuinness D, Patel-Schneider P, Nardi D (eds) (2003) The description logic handbook: Theory, implementation and applications. Cambridge University Press

Basic BD, Snajder J (2019) Lecture 7: logic programming in prolog. University of Zagreb Artificial Intelligence Lecture Slides. https://www.fer.unizg.hr/_download/repository/AI-7-LogicProgramming.pdf

Buchanan BG, Shortliffe EH (1984) Rule-based expert systems: the MYCIN experiments of the Stanford heuristic programming project (the Addison-Wesley series in artificial intelligence). Addison-Wesley Longman Publishing Co., Inc.

Cochez M, Decker S, Prud'Hommeaux E (2016) Knowledge representation on the web revisited: the case for prototypes. In: The Semantic Web–ISWC 2016: 15th international semantic web conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I 15, Springer, pp 151–166
Coppin B (2004) Artificial intelligence illuminated. Jones & Bartlett Learning
Ernst GW, Newell AG (1969) A case study in generality and problem-solving. Academic, New York
Grimm S, Hitzler P, Abecker A (2007) Logic, ontologies and semantic web languages. In: Studer R, Grimm S, Abecker A (eds) Semantic web services: concepts, technologies, and applications. Springer
Gruber TR (1995) Toward principles for the design of ontologies used for knowledge sharing? International Journal of Human-computer Studies 43(5-6):907–928
Jackson P (1998) Introduction to expert systems, 3rd edn. Addison Wesley
James P (1992) Knowledge graphs. In: 1991 Workshop on linguistic instruments in knowledge engineering, Tilburg, January 18–19, Elsevier, pp 97–117
Kant I (1783) Prolegomena zu einer jeden künftigen Metaphysik die als Wissenschaft wird auftreten können. Project Gutenberg. https://www.projekt-gutenberg.org/kant/prolegom/prolegom.html
Lenat DB, Feigenbaum EA (1987) On the thresholds of knowledge. In: Proceedings of the tenth international joint conference on artificial intelligence, August, pp 23–28
Martinez M, Rohrer E, Severi P (2016) Complexity of the description logic ALCM. In: Proceedings of the fifteenth international conference on principles of knowledge representation and reasoning, Cape Town South Africa, April 26–29, AAAI Press, KR'16, pp 585–588
McCarthy J, Minsky ML, Rochester N, Shannon CE (2006) A proposal for the Dartmouth summer research project on artificial intelligence, August 31, 1955. AI Mag 27(4):12–12
Minsky M (1975) A framework for representing knowledge. In: Winston PH (ed) The psychology of computer vision. McGraw-Hill, New York, pp 211–277
Newell A, Shaw J (1959) A variety of intelligent learning in a general problem solver. RAND Report P-1742, July 6
Newell A, Simon HA (1972) Human problem solving, vol 104. Prentice-Hall, Englewood Cliffs, NJ
Newell A, Shaw JC, Simon HA (1959) Report on a general problem-solving program. In: Proceedings of the first international conference on information processing congress, UNESCO, Paris, June 15–20, Pittsburgh, PA, pp 256–264
Ortiz M (2010) Lecture 5: Complexity of reasoning in ALC. Declarative knowledge processing lecture slides. http://www.kr.tuwien.ac.at/education/deklslides/ws12/dkpWS10 6.1nup.pdf
Rector AL, Nowlan WA, & Galen Consortium (1994) The GALEN project. Comput Methods Prog Biomed 45(1–2):75–78
Rodriguez ND (2019) Introduction to description logics and ontologies. Symbolic artificial intelligence lecture slides. https://perso.telecomparitech.fr/bloch/OptionIA/IA301-Lecture1-IntroDL-OWL.pdf
Schreiber G, Wielinga B, de Hoog R, Akkermans H, Van de Velde W (1994) CommonKADS: a comprehensive methodology for KBS development. IEEE Expert 9(6):28–37
Shannon CE (1950) Programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 41(314):256–227
Simon HA (1957) Models of man: social and rational-mathematical essays on rational human behavior in a social setting. Wiley
Sowa JF (ed) (1991) Principles of semantic networks: explorations in the representation of knowledge. Morgan Kaufmann
Studer R, Benjamins VR, Fensel D (1998) Knowledge engineering: principles and methods. Data Knowl Eng 25(1-2):161–197

Van Melle W (1978) MYCIN: a knowledge-based consultation program for infectious disease diagnosis. International Journal of Man-Machine Studies 10(3):313–322

Wielinga BJ, Schreiber AT, Breuker JA (1992) KADS: a modelling approach to knowledge engineering. Knowl Acquis 4(1):5–53

Williams A (2005) Prolog, resolution and logic programming. CS612 automated reasoning lecture slides. http://www.cs.man.ac.uk/schmidt/CS612/2005-2006/resolution-slides.pdf

Chapter 3
Information Retrieval and Hypertext



Information retrieval (IR) is about finding material (documents, graphics, voice, video) of an unstructured nature that satisfies an information need from within large collections [see Manning et al. (2008), Teufel (2014a), and Teufel (2014c)] that are usually stored on computers. In this chapter, we focus on the retrieval of text. Searching for information in human storage systems has always been a challenge for humanity. It goes as far as the third century BC and earlier. The main challenge is how do I find the relevant information sources given a query?

Information retrieval research led to the development of various methods and techniques for representing queries and documents and finding the most relevant documents:

- *Boolean models* consider documents as a set of words.
- *Vector space models* consider documents as word vectors.
- *Probabilistic models* consider the similarity of a query to a document as probabilities. This can be used as part of the Boolean and vector space models.

We will give a bit more detail about the Boolean model, the vector space models, and the PageRank algorithm used on the Web. We add a few applications and discuss the specific area of hypertext-based systems, given their relevance for the Web.

3.1 Boolean Model

The initial work on IR focused on converting user queries to Boolean expressions and looking for exact matches among the *set* of words comprising each document. Consider the following query as an example: Which plays of Shakespeare are about Brutus and Caesar but not Calpurnia (Teufel 2014a)? The main idea is to record for each play whether it contains these words (see Fig. 3.1).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_3

Query: Brutus AND Caesar AND NOT Calpurnia

Word \ Play	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
Antony	1	1	0	0	0
Brutus	**1**	1	0	**1**	0
Caesar	**1**	1	0	**1**	1
NOT Calpurnia	**1**	0	1	**1**	1
Cleopatra	1	0	0	0	0
mercy	1	0	1	1	1
Query	**1**	0	0	**1**	0

Fig. 3.1 An excerpt from the Boolean representation of unique words in Shakespeare's plays, adapted from Manning et al. (2008) and Teufel (2014a)

The Boolean model was mainly used by large commercial information providers until the 1990s, e.g., IR systems for lawyers. The main advantages of the Boolean model are straightforward implementation and knowing exactly what you will get since a word either matches the document or does not match. The disadvantages are that the results are not ranked in any way and the term weights are not considered. The approach is also semantically highly brittle as they cannot handle, for example, synonyms or negation. For example, a query to retrieve the documents about the word "birds" would retrieve a document containing the sentence "This document is not about birds."

Therefore, extended Boolean models were developed to address some shortcomings (Teufel 2014a). For example, a proximity operator is introduced. This means a document is matched only if some of the words in the query appear in the document at a given proximity (Manning et al. 2008). Also, some implementations of the Boolean model allowed weighting of the terms in the query and provided fuzzy matching. For example, in a conjunctive query, the document that does not contain all words still receives a matching score based on the weights of the terms in the query.[1]

3.2 Vector Space Model

The standard Boolean model returns the documents matching the query exactly. What happens if there are 1000 documents matching a query? Results are unordered: a user must go through all documents. What happens if there are no exact matches?

[1] See also: https://courses.cs.vt.edu/~cs5604/cs5604cnIF/IF3.html

$$similarity = \cos(\theta) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Fig. 3.2 Vector-based calculation of similarity

Play / Word	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
Antony	157	73	0	0	0
Brutus	4	157	0	2	0
Caesar	232	227	0	2	1
Calpurnia	0	10	0	0	0
Cleopatra	57	0	0	0	0
mercy	2	0	3	8	5

Fig. 3.3 An excerpt from the representation of unique words in Shakespeare's plays as term frequency vectors, adapted from Teufel (2014b)

Maybe there are some documents that match to some extent. The vector space model allows representing *documents* and *queries* as *N*-dimensional vectors (N = number of unique words in a document). The main idea is to calculate the similarity between two vectors (Teufel 2014b). Cosine similarity[2] is a good way to calculate the degree of overlap between two vectors. It is based on calculating the angle between two vectors in the vector space. *A* and *B* in Fig. 3.2 are two vectors representing the query and a potential answer.

The most important choice is to decide how to represent documents as vectors. Binary incidence vectors represent documents in a way similar to the Boolean method; however, this does not provide a way to rank terms in a document. What we can do is assign a weight ($w_{t,\,d}$) for each word (t) for each document (d). Term frequency uses the frequency of a word in a document instead of binary values (Fig. 3.3). We say $tf_{t,\,d}$ is the term frequency of term t in document d, and it is defined as the number of times that t occurs in d (Teufel 2014b).

Raw term frequency is a start but not the best metric for calculating $w_{t,\,d}$ to build vectors. A document with $tf = 100$ of a term is more relevant than a document with $tf = 1$ of that term but not 100 times more relevant. Relevance does not depend proportionally on term frequency. Therefore, log frequency is used instead to alleviate this effect of proportion (Teufel 2014b):[3]

[2] https://en.wikipedia.org/wiki/Cosine_similarity

[3] The addition of "1" is a normalization step to prevent log frequency from producing zero for the words that occur only once in a document since log1 would be zero.

Word \ Play	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
Antony	5.25	3.18	0	0	0
Brutus	1.21	6.10	0	1.0	0
Caesar	8.59	2.54	0	1.51	0.25
Calpurnia	0	1.54	0	0	0
Cleopatra	2.85	0	0	0	0
mercy	1.51	0	1.90	0.12	5.25

Fig. 3.4 An excerpt from the TF-IDF vector representation of the words in Shakespeare's plays, adapted from Teufel (2014b)

$$w_{t,d} = \begin{cases} 1 + \log tf_{t,d} & \text{if } tf_{t,d} > 0 \\ 0 & otherwise \end{cases}$$

Most frequently occurring words do not necessarily distinguish documents in terms of relevance to a query. Take the word "the" as an example, which occurs in practically every English document. Rare words are more significant for deciding the relevance of a document to a query. Therefore, we want to weigh rare terms more than frequent terms across documents. In addition to the term frequency, we also want to use the *inverse document frequency* of these terms. If a word appears in a document frequently but rarely in the collection of documents, it indicates that this document is more relevant for that term. The inverse document frequency of a term t (idf_t) is calculated as follows (Teufel 2014b). Let us say that df_t is the document frequency of t, i.e., the number of documents in which t occurs, and N is the number of documents in the collection. Then idf_t is calculated as:

$$idf_t = \log \frac{N}{df_t}$$

The *log* of the ratio is used to alleviate the proportion effect, as we discussed when we talked about term frequency. $w_{t,\,d}$ based on TF-IDF is then calculated as follows:

$$w_{t,d} = \begin{cases} (1 + \log tf_{t,d}) * idf_t & if\ tf_{t,d} > 0 \\ 0 & otherwise \end{cases}$$

Note that intuitively, there are two ways the weight of a term for a document can be 0. Either the word does not appear in that document (i.e., its term frequency is 0), or the word appears in every document (i.e., its logarithmic inverse document frequency is 0). Figure 3.4 shows an excerpt from the unique words that occur in

Shakespeare's plays. The words are represented as vectors where each vector contains the TF-IDF weights of the corresponding word.

3.3 Evaluation of Information Retrieval Systems

Information retrieval systems are evaluated by means such as *precision*, *recall*, and *F-score*. Precision and recall[4] are the most common measures for evaluating IR systems (Teufel 2014c). Precision is the ratio of the number of retrieved documents that are relevant to the query to the number of retrieved documents. The following terminology is typically used while calculating precision, recall, and F-score:

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}$$

Recall is the ratio of the number of retrieved documents that are relevant to the query to the number of all relevant documents:

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|}$$

F-score[5] is the harmonic mean of precision and recall evaluating the effectiveness of an IR system:

$$F = 2 * \frac{precision * recall}{precision + recall}$$

Note that there is a trade-off between precision and recall (Fig. 3.5). A system can simply return all documents to have 100% recall; however, the precision will be low. In contrast, high precision means that recall will be low.

For IR at a large scale, for example, in Web search, precision and recall may be impractical measures. Who cares about general precision if the search engine returns 1,000,000 results? Users are generally interested in just checking a dozen results. What is the precision among the top 10 results? The importance of ranking increases significantly: try to get the top k results as relevant as possible to the user's information need. Originally, the query-dependent ranking was dominant, e.g., *TF-IDF* vector similarity between query and document. Meanwhile, a prominent alternative has become a query-independent ranking, e.g., the PageRank algorithm.

[4] https://en.wikipedia.org/wiki/Precision_and_recall - distributed under CC-BY-SA 3.0

[5] https://en.wikipedia.org/wiki/F-score

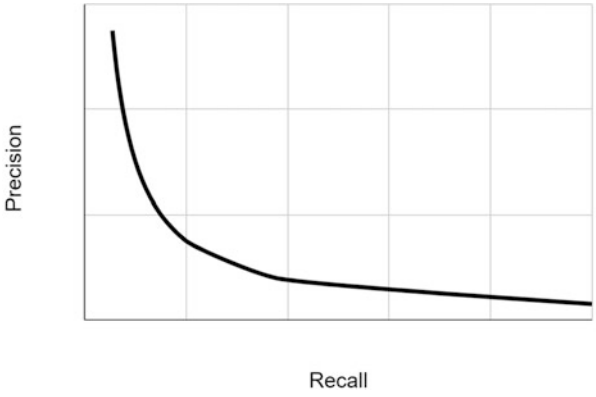


Fig. 3.5 Trade-off between precision and recall

3.4 PageRank

PageRank[6] (Page et al. 1999) has been an algorithm used by Google for many years since its founding. Its main goal is to identify how important a Web page is.

The importance of a Web page is identified mainly based on the incoming links from other pages. Pages "vote" for the importance of a page by providing links to it. How much of a vote is given by a page depends on the page rank of that page. Actually, its value is recursively defined by the Web pages that point to it. And how are their values defined? It is by the value of the Web pages that point to each of those Web pages and so on. To be precise, a Web page distributes its value on all the Web pages it points to. That is, if it points to five Web pages, then each of them receives 20% of its value. So even Web pages with a high value through ingoing links may give only little value to their outgoing links when there are many of them. The calculation must happen recursively: but the Web is huge; where and when do we stop with the iterations? However, at some point, the changes in the calculated page rank value will get very small; then we can stop the calculation.

Assume page A has incoming links from pages $T_1 \ldots T_n$. Parameter d is a damping factor that can be assigned a value between 0 and 1 (usually 0.85).[7] $PR(T_i)$ is the page rank of the ith page that has an outgoing link to A. $C(T_i)$ is the total number of outgoing links from the ith page. The PageRank of page A is calculated as follows (Brin and Page 1998)[8]:

[6] http://ianrogers.uk/google-page-rank/.

[7] "PageRank can be thought of as a model of user behavior. We assume there is a 'random surfer' who is given a web page at random and keeps clicking on links, never hitting 'back' but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank. And the d damping factor is the probability at each page the 'random surfer' will get bored and request another random page"—(Brin and Page 1998).

[8] See also http://ianrogers.uk/google-page-rank/ for a more detailed explanation and example.

$$\begin{pmatrix} L_{11} & \cdots & L_{1n} \\ \vdots & \ddots & \vdots \\ L_{n1} & \cdots & L_{nn} \end{pmatrix} * \begin{pmatrix} r_1^i \\ \vdots \\ r_n^i \end{pmatrix} = \begin{pmatrix} r_1^{i+1} \\ \vdots \\ r_n^{i+1} \end{pmatrix} \rightarrow \begin{pmatrix} \sum_{j=1}^{n} L_{1j} * r_1^i \\ \vdots \\ \sum_{j=1}^{n} L_{nj} * r_n^i \end{pmatrix} = \begin{pmatrix} r_1^{i+1} \\ \vdots \\ r_n^{i+1} \end{pmatrix}$$

Fig. 3.6 Iterative eigenvector computation

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)}\right)$$

At the first iteration, the PageRank of each page can be initialized as $\mathrm{PR}(T_1) = \mathrm{PR}(T_2) = \mathrm{PR}(T_n) = 1$. The PageRank values of each page are updated after each iteration until they do not change (or change very little).

The iterative calculation process becomes clearer from a linear algebraic perspective. The PageRank calculation is simply finding an *eigenvector r* of a matrix *L* that represents the links between the Web pages on the Web:[9]

$$L * r = r$$

As the formula shows, the calculation is self-referential and must be solved iteratively. Figure 3.6 shows the iterative computation of an eigenvector r of matrix L.

Now, consider a subset of the Web that consists of pages *A*, *B*, *C*, and *D*, as shown in Fig. 3.7.

We can represent this Web as an $n \times n$ link matrix (L). Each column represents a page and its outgoing links to another page, while rows represent the incoming links.

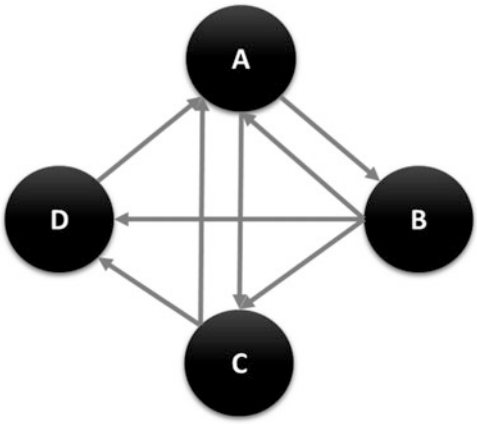


Fig. 3.7 A small Web of documents

[9] For simplicity, we ignore the damping factor *d*.

$$L = \begin{pmatrix} 0 & 0.33 & 0.5 & 1 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0 & 0.33 & 0.5 & 0 \end{pmatrix} \qquad r^0 = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}$$

Fig. 3.8 The small Web of documents represented as the link matrix L and r^0 rank vector initialized

$$r^1 = \begin{pmatrix} 0 & 0.33 & 0.5 & 1 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0 & 0.33 & 0.5 & 0 \end{pmatrix} * \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix} = \begin{pmatrix} 0.46 \\ 0.13 \\ 0.21 \\ 0.21 \end{pmatrix}$$

$$r^2 = \begin{pmatrix} 0 & 0.33 & 0.5 & 1 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0 & 0.33 & 0.5 & 0 \end{pmatrix} * \begin{pmatrix} 0.46 \\ 0.13 \\ 0.21 \\ 0.21 \end{pmatrix} = \begin{pmatrix} 0.35 \\ 0.23 \\ 0.27 \\ 0.15 \end{pmatrix}$$

…

$$\boldsymbol{r^{10}} = \begin{pmatrix} 0 & 0.33 & 0.5 & 1 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0 & 0.33 & 0.5 & 0 \end{pmatrix} * \begin{pmatrix} 0.37 \\ 0.18 \\ 0.24 \\ 0.19 \end{pmatrix} = \begin{pmatrix} 0.37 \\ 0.18 \\ 0.24 \\ 0.18 \end{pmatrix}$$

$$\boldsymbol{r^{11}} = \begin{pmatrix} 0 & 0.33 & 0.5 & 1 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0 & 0.33 & 0.5 & 0 \end{pmatrix} * \begin{pmatrix} 0.37 \\ 0.18 \\ 0.24 \\ 0.18 \end{pmatrix} = \begin{pmatrix} 0.37 \\ 0.18 \\ 0.25 \\ 0.18 \end{pmatrix}$$

Fig. 3.9 The iterative eigenvector calculation process

The values are normalized by the total number of outgoing links from a page to convert them into probabilities of a user reaching a page from another page. For example, L_{21} is the probability of a user reaching from *A to B*, and L_{31} represents the probability of a user reaching from *A to C*.

Given a link matrix L and a rank vector r^0 (each row corresponds to the rank of pages *A to D*), we will calculate the final rank vector r that contains the ranking for the Web pages *A to D*. See Fig. 3.8 for the initial stage of the calculation.

The tricky part is that at the beginning, we do not know the rank vector r. So the initial step of the iteration starts with $1/n$ as the rank of each page, where n is the number of pages. In our example, r would be a 4×1 vector with 0.25 at each row ($n = 4$).

Figure 3.9 demonstrates the calculation of the vector *r*. We make the matrix multiplication iteratively until it converges to the desired eigenvector. At each step, we use the rank vector obtained from the previous stage. We know we found that vector when *r* stops changing at each iteration. As shown in Fig. 3.9, the calculation converges around the 10th–11th iteration, which indicates that we found vector *r*. Note that the actual values in the vector is not particularly important, but it is how they are ordered in terms of magnitude. In our example, the final vector *r* obtained from the 11th iteration shows that the highest ranked page is *A*, followed by *C*, which is followed by equally ranked pages *B* and *D*. Obviously, one can ask many questions:

- Is there always an eigenvector?
- Is the eigenvector unique?
- Does the iteration always converge?
- What is the mathematical role of the damping factor?

However, we would need to add more on linear algebra, which is beyond our scope.[10]

3.5 Applications and Systems

The explosion of the information available toward the second half of the twentieth century motivated people to think about effective and efficient methods of information retrieval (Sanderson and Croft 2012). The first computerized machine for information retrieval was invented in 1948 (Holmstrom 1948) and was implemented with a Univac computer. It searches for text references on magnetic steel tape, given a subject code as a query. It was able to process 120 words per minute. A breakthrough for IR happened toward the end of the twentieth century with the invention of the Internet and the Web. Search engines appeared as large-scale IR systems on the Web. Early search engines crawled a (significant) portion of the Web, created indexes of the crawled pages, and retrieved documents (i.e., Web pages) given a query.

3.6 Hypertext

The theoretical *Memex* machine from Vannevar Bush (1945) is considered the inspiration for hypertext systems. The Memex machine is a large desk with a mechanism that can store large information sources in microfilms. The user can

[10] We recommend watching https://www.coursera.org/lecture/linear-algebra-machine-learning/introduction-to-pagerank-hExUC and https://www.youtube.com/watch?v=-RdOwhmqP5s

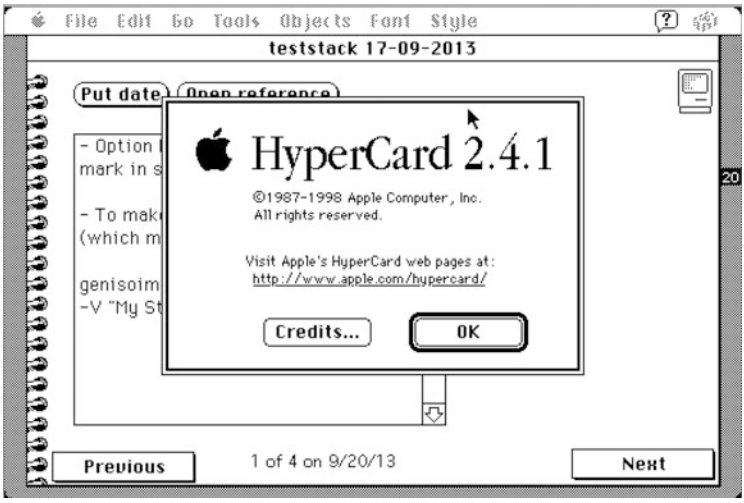


Fig. 3.10 The interface of HyperCard (Image by 37 Hz, distributed under CC BY 2.0 license. https://www.flickr.com/photos/37hz/9842400043)

enter a code and retrieve the relevant microfilm magnified on a translucent screen. The system also allows linking content on microfilms to create "trails of information."

Inspired by the Memex idea, in 1962, Doug Engelbart started to work on text processing software (oN-Line System (NLS)) that has the capability of linking other peoples' work within a text. The term hypertext was coined and later expanded to hypermedia by Ted Nelson in 1965 (Rayward 1994; Nielsen 1995). In the 1960s, 1970s, and 1980s, many other hypertext systems were developed (see Nielsen (1995) for a comprehensive list). Among those, *HyperCard* was one of the most successful commercial hypermedia products (Fig. 3.10). It was developed by Bill Atkinson from Apple in 1987. The concept of the system is based on a stack of virtual cards. Each card can contain text enhanced with links and other interactive elements, like textboxes, checkboxes, and buttons. Documents can be written in a language called Hypertalk, which also provides a graphical development interface. Programming with Hypertalk was somewhat similar to form-based application development with Visual Basic.

These concepts and techniques got a significant increase in importance with the World Wide Web, which opened these closed systems and connected them via the Internet.

3.7 Summary

Information retrieval (IR) deals with the task of retrieving relevant documents from a collection of documents given a query. The development of IR systems was an answer to the information explosion starting in the 1940s. At the core of IR systems lies deciding "to what extent a document is relevant for a query." Initial approaches used a Boolean model. With this model, only the documents that match the entire query were retrieved without any indication of the level of relevance (i.e., ranking). Various extensions of the Boolean model and approaches using the vector space model and probabilistic models allowed incorporating a ranking mechanism. For example, approaches using the vector space model could tell how relevant a document to a query is by measuring the cosine distance between the query and document vectors. IR found many commercial applications in the 1990s in domains like banking and law.

Toward the end of the twentieth century, the invention of the World Wide Web brought a new challenge to IR: the collection of documents was now much larger and distributed. Moreover, the documents were linked to each other as the Web is built on hypertext. The proper ranking of relevant documents became more important as the users were interested in a small number of the most relevant and important pages across the large network of documents. Google developed the PageRank algorithm that benefits from the principles of hypertext. Simply put, the more a page was linked from other pages, the higher the rank it was assigned.

Information retrieval, particularly in the context of the Web, undoubtedly changed our lives. As discussed in Chap. 1, traditional IR methods worked well for many years for search engines; however, they are almost purely syntactic, which means they are susceptible to errors when negation and synonyms are involved. IR supported with knowledge graphs opened new doors for users. The queries were not only matched with documents but also with the actual information they were looking for.

References

Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems 30(1–7):107–117

Bush V (1945) As we may think. The Atlantic Monthly 176(1):101–108

Holmstrom J (1948) Section III. Opening plenary session. In: The royal society scientific information conference, June 21–July 2, London

Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press

Nielsen J (1995) Multimedia and hypertext: the Internet and beyond. Morgan Kaufmann

Page L, Brin S, Motwani R, Winograd T (1999) The PageRank citation ranking: bringing order to the web. Technical reports, Stanford InfoLab

Rayward WB (1994) Visions of Xanadu: Paul Otlet (1868–1944) and hypertext. J Am Soc Inf Sci 45(4):235–250

Sanderson M, Croft WB (2012) The history of information retrieval research. Proceedings of the IEEE 100 (Special Centennial Issue), pp 1444–1451. doi:https://doi.org/10.1109/JPROC.2012.2189916

Teufel S (2014a) Introduction and overview. Information retrieval. https://www.cl.cam.ac.uk/teaching/1314/InfoRtrv/lecture1.pdf

Teufel S (2014b) Term weighting and the vector space model. URL https://www.cl.cam.ac.uk/teaching/1314/InfoRtrv/lecture4.pdf

Teufel S (2014c) Evaluation. https://www.cl.cam.ac.uk/teaching/1314/InfoRtrv/lecture5.pdf

Chapter 4
The Internet



The Internet is one of the things the Cold War brought to humanity. At that time, the Soviets were shooting Sputnik into space, and the USA did not want to stay behind in technological advances. So they founded the Advanced Research Projects Agency (ARPA) in the 1960s. At that time, ARPA started to support work on the communication of computers on large networks. This research led to the Advanced Research Projects Agency Network (ARPANET). The first communication was done between a computer at Menlo Park and the University of California, Los Angeles. It was a node-to-node communication to transfer the message "LOGIN" between two computers.[1]

The initial way of connecting different physical networks was problematic. They had rigid routing structures and were fragile as they had a single point of failure. Both American and English researchers were meanwhile working on a more robust way of transferring data on a network. In the end, the concept of *packet switching* was adopted. Packet switching allows messages to be transported as *small packages* in arbitrary order over more flexible routes[2] (Fig. 4.1).

First, there is the *Internet Protocol (IP)*,[3] which provides the network layer communication protocol of the Internet. Its task is to deliver the data packages across various networks based on IP addresses in the header of the data and encapsulate the data to be delivered.

On top of IP (responsible for the lower layer transfer of packages and containing the IP addresses) is the higher-level transport protocol *TCP*.[4] Soon, there were many application-level protocols developed, for example:

[1] https://www.bbc.com/news/business-49842681

[2] https://en.wikipedia.org/wiki/History_of_the_Internet

[3] https://en.wikipedia.org/wiki/Internet_Protocol licensed under CC-BY-SA 3.0.

[4] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_4

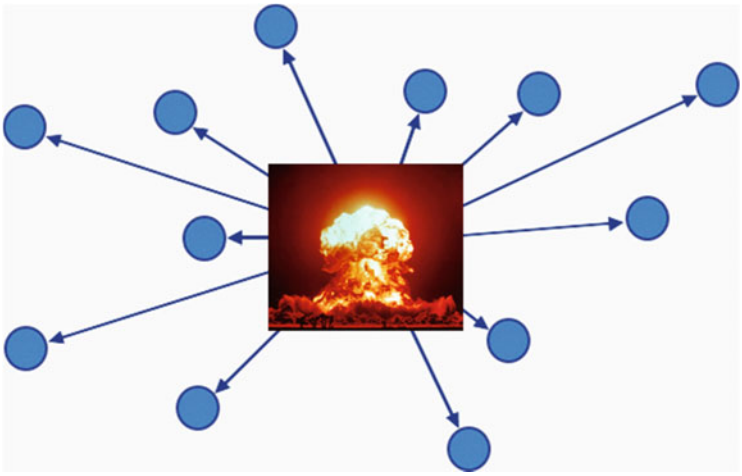


Fig. 4.1 Prevent the central point of failure! (The nuclear explosion photo is the courtesy of the National Nuclear Security Administration Nevada Field Office. Licensed under Public Domain.)

- *File Transfer Protocol (FTP):* transferring files between server and client.
- *Domain Name System (DNS):* a naming system that maps human-friendly domain names to IP addresses.
- *Simple Mail Transfer Protocol (SMTP)* provides a protocol for e-mail communication.
- *Hypertext Transfer Protocol (HTTP):* the Internet protocol for exchanging hypertext objects.

The Internet connects the edges retaining no state and aims for speed and simplicity. To that end, an important principle is the robustness principle:

> In general, an implementation should be conservative in its sending behavior, and liberal in its receiving behavior. That is, it should be careful to send well-formed datagrams, but should accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear). – Internet Protocol Specification, August 1979[5]

The Internet uses the *Open System Interconnection (OSI) model*. The OSI model layers are (Fig. 4.2):[6]

1. The *physical layer* provides the physical interface between the device and the transmission media.
2. The *data link layer* provides the transmission protocol controlling the data flow between network devices.

[5] https://www.postel.org/ien/txt/ien111.txt

[6] https://insights.profitap.com/osi-7-layers-explained-the-easy-way

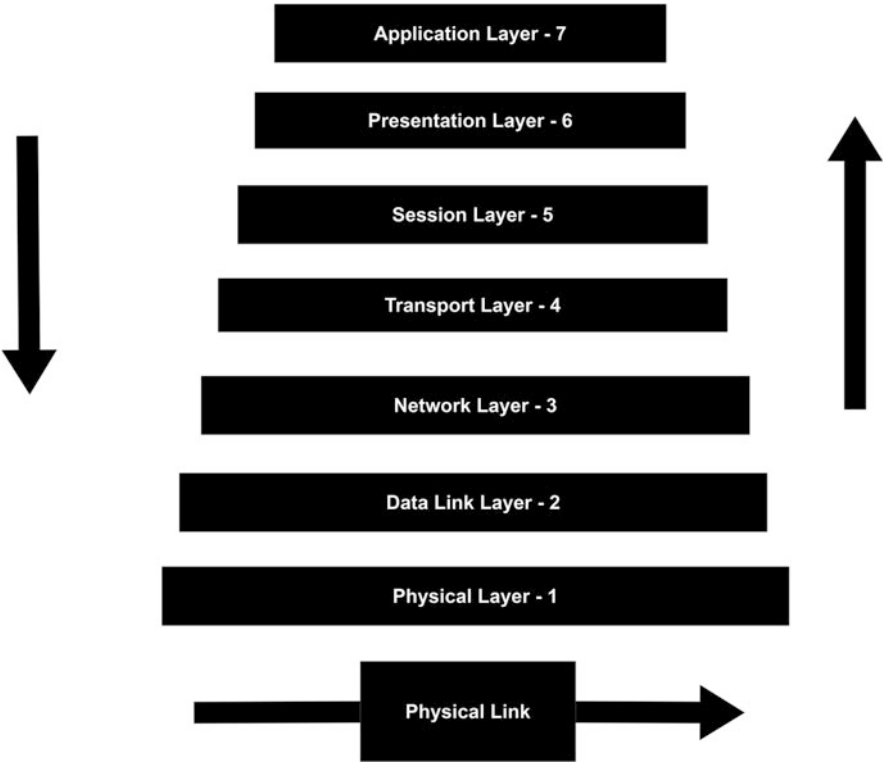


Fig. 4.2 The OSI layer model

3. The *network layer* provides the necessary routing and switching technologies. It routes data packets of variable length from a source to a destination network. This is provided by the Internet protocol (IP).
4. The *transport layer* transfers data between end users, ensuring that the data transfer is error- and congestion free. This is provided by the Transmission Control Protocol (TCP).
5. The *session layer* takes care of the management, establishment, and termination of connections between two end users of a network.
6. The *presentation layer* translates data for the application layer. It also takes care of encryption and authentication.
7. Finally, the *application layer* provides functionality for applications.

The main standards used by the Internet are Unicode, IP, and TCP. In the following, we will briefly introduce these standards.

Unicode standard provides an international character set, e.g., that is used to encode the data in the data package.

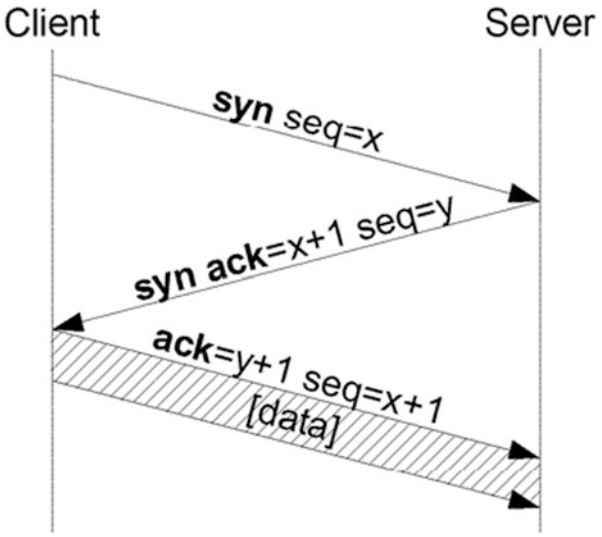


Fig. 4.3 Flags during a TCP handshake between client and server (Image by Snubcube, distributed under CC-BY-SA 3.0 https://de.wikipedia.org/wiki/Datei:Tcp-handshake.svg)

The Internet Protocol (IP)[7] is the network layer communication protocol, e.g., that enables internetworking and establishes the Internet. Its routing function enables internetworking, which establishes the Internet. It delivers packets from source to destination based on IP addresses in packet headers.

The Transmission Control Protocol (TCP)[8] is one of the main protocols of the Internet, enabling applications to exchange messages over a network. It is part of the Internet Protocol Suite (TCP/IP) located at the transport layer (layer 4). TCP enables reliable, ordered, and error-checked transmission of a stream of bytes between applications running on an IP network. TCP is connection oriented, i.e., a connection between client and server must be built before data can be sent. Flags are important for connection handling (Fig. 4.3):[9]

- *ACK*—indicates that the Acknowledgment field is significant.
- *FIN*—last packet from the sender.
- *SYN*—synchronize sequence numbers; only the first packet sent from each end should have this flag set.

Based on this, a client can establish a connection with a server via a TCP handshake as follows:[10]

[7] Reprinted from https://en.wikipedia.org/wiki/Internet_Protocol licensed under CC-BY-SA 3.0.

[8] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[9] Reprinted from https://en.wikipedia.org/wiki/Transmission_Control_Protocol#TCP_segment_structure licensed under CC-BY-SA 3.0.

[10] Reprinted from https://en.wikipedia.org/wiki/Transmission_Control_Protocol#Connection_establishment licensed under CC-BY-SA 3.0.

1. *SYN*: the active open is performed by the client sending an SYN to the server. The client sets the segment's sequence number to a random value, x.
2. *SYN-ACK*: in response, the server replies with an SYN-ACK. The acknowledgment number is set to one more than the received sequence number, i.e., $x + 1$, and the sequence number that the server chooses for the packet is another random number, y.
3. *ACK*: finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgment value, i.e., $x + 1$, and the acknowledgment number is set to one more than the received sequence number, i.e., $y + 1$.

Even though development and progress on the Internet are achieved in a decentralized way, there is a need for governance and standardization. The organizations are the Internet Engineering Task Force (IETF), the Internet Corporation for Assigned Names and Numbers (ICANN), and the Internet Assigned Numbers Authority (IANA).

Chapter 5
The World Wide Web



Many people use the Internet and WWW as synonyms. However, the Internet is an infrastructure to interconnect various heterogeneous networks, and the Web is one of the most important application layers on top of this infrastructure, the Internet provides.

The *World Wide Web* (*WWW* or simply the *Web*) is "a system of interlinked, hypertext documents that runs over the Internet. With a Web browser, a user views Web pages that may contain text, images, and other multimedia and navigates between them using hyperlinks."[1] Its essence is easy to understand: it combines a hypertext infrastructure with the Internet.

The end of the 1980s and the beginning of the 1990s marked an important point that shaped developments in artificial intelligence (AI), the Internet, databases, information retrieval, hypertext, and natural language processing. The World Wide Web was invented as a combination of Internet and hypertext systems by Sir Tim Berners-Lee. The amount of content, data, and services published on the Web rapidly came to an enormous size within the following decades. In the Web, links can refer to content stored at external computers and systems in extension to traditional hypertext systems. The Web is built on many technologies, but the following are arguably at the center:

- A common naming and addressing schema: Uniform Resource Identifiers (URIs) described in Request for Comments (RFC) 3986.[2]
- A common protocol for accessing resources and exchanging their various representations based on the REST principles: Hypertext Transfer Protocol (HTTP) described in RFC 2616.[3]
- A markup language for allowing applications like Web browsers to render hypermedia content: Hypertext Markup Language (HTML). The first versions

[1] http://en.wikipedia.org/wiki/World_Wide_Web

[2] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

[3] https://www.rfc-editor.org/rfc/rfc9110.html

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_5

were described in RFC 1866. Later versions were standardized by W3C and enriched toward XML.[4]

All these technologies are standardized by bodies like the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). "The World Wide Web Consortium (W3C) is an international community where Member organizations, a full-time staff, and the public work together to develop Web standards."[5] The W3C aims for the worldwide availability of Web access and envisions Web progress in terms of interaction, data, services, and security through standardization. Comparable to IETF's standardization through RFCs, W3C publishes reports that pass different maturity levels until they are officially recommended:

1. Working Draft (WD)
2. Candidate Recommendation (CR)
3. Proposed Recommendation (PR)
4. W3C Recommendation (REC)

5.1 Uniform Resource Identifiers (URIs)

The Web assumes that resources are anything we want to talk about. Uniform Resource Identifiers (URIs) denote ('are names for') these resources. Obviously, it is necessary to distinguish between the name of a thing (URI) and the thing itself (resource) and its representation (Fig. 5.1).

A Uniform Resource Identifier (URI) is a string of characters used to identify a resource on the Internet. Such a URI can be a URL or a URN (see Fig. 5.2).

A *Uniform Resource Name (URN)* defines an item's identity: the URN *urn:isbn:0-395-36,341-1* is a URI that specifies the identifier system, i.e., the International Standard Book Number (ISBN), as well as the unique reference within that system, and allows one to talk about a book but does not suggest where and how to obtain an actual copy of it.

A *Uniform Resource Locator (URL)* provides a location and a method for finding the resource. For example, the URL *http://www.sti-innsbruck.at/* identifies a resource (STI's home page) and implies that a representation of that resource (such as the home page's current HTML code as encoded characters) is obtainable via HTTP from a network host named www.sti-innsbruck.at.[6]

The URI syntax has the following high-level structure:[7]

```
URI = scheme ":" path [ "?" query ] [ "#" fragment ]
```

[4] https://www.w3.org/XML/

[5] https://www.w3.org/Consortium

[6] Adapted from https://wiki.eclipse.org/File_URI_Slashes_issue

[7] https://www.ietf.org/rfc/rfc3986.txt

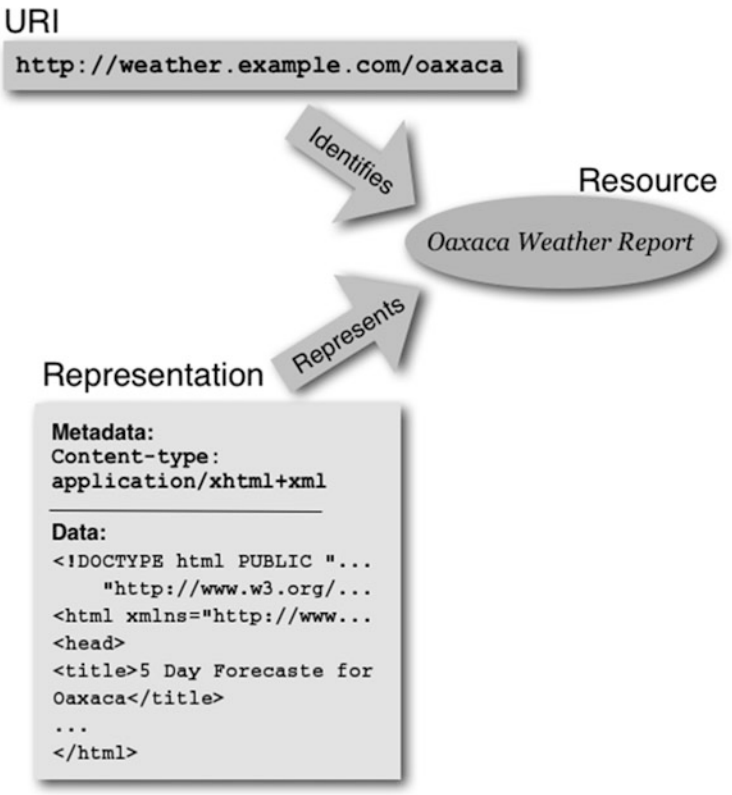


Fig. 5.1 The distinction between a URI, the thing itself, and its representation (https://www.w3.org/wiki/HttpUrisAreExpensive Copyright © 2010 World Wide Web Consortium. https://www.w3.org/Consortium/Legal/2023/doc-license)

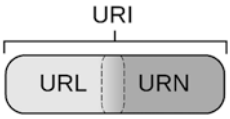


Fig. 5.2 URIs, URLs, and URN and their relationship (Image by David Torres, published under CC-BY-SA 3.0 license https://commons.wikimedia.org/wiki/File:URI_Euler_Diagram_no_lone_URIs.svg)

5.2 REST and HTTP

The Web is based on a simple and robust architecture, i.e., the *Representational State Transfer (REST)*, which is the theoretical foundation for Web architecture principles (Fielding 2000). Requests and responses are based on the transfer of "representations" of "resources." A simple, stateless, and uniform protocol to access information chunks, i.e., the *Hypertext Transfer Protocol (HTTP)* for client/server

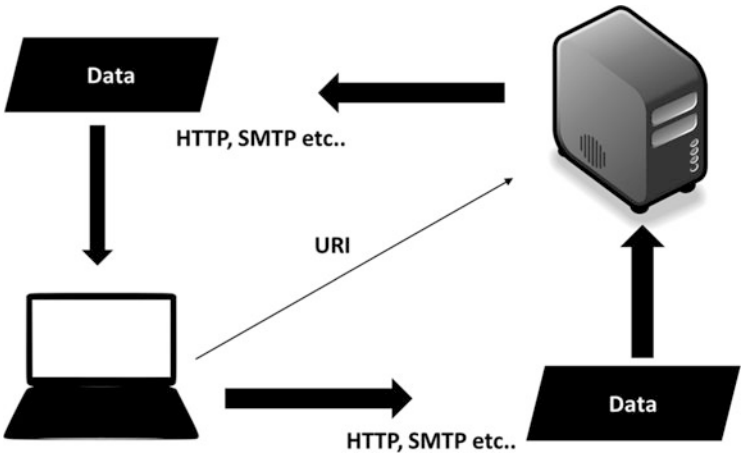


Fig. 5.3 The architectural style of the Internet and the Web (Image adapted from https://www.w3.org/People/Frystyk/thesis/WWW.html)

communication, implements this architecture. Therefore, HTTP is a good example of a REST-style implementation; however, it should not be confused with REST. REST is a style of software architecture for distributed systems such as the World Wide Web providing a client/server architectural style (Fig. 5.3).[8] Requirements supported by REST-enabled systems stem from the requirements addressed by any system following the Web architecture (Fielding and Taylor 2002):

- *Simplicity*: low-entry barrier, rapid Web Application Programming Interface (API) adoption
- *Extensibility*: fostering easy growth.
- *Distributed hypermedia*: depends on the well-established notions of hypermedia.
- *Scalability at the Web level*: should rely on technologies/protocols that allow scaling on a large distributed system like the Web.
- *Independent deployment*: old and new solutions must be able to exist together.

Information is organized in the form of resources that correspond to specific information and are referenced with a global identifier (e.g., a URI in HTTP). Components of the network (user agents and origin servers) communicate via a standardized protocol (e.g., HTTP) by exchanging resource representations (e.g., a JSON object representing the resource). Any number of connectors (e.g., clients, servers, caches, tunnels, etc.) can mediate the request, but each does so without knowing anything other than their own request. An application only needs to know the following about a resource to interact with it: its identifier, representation formats, and operation needed to be taken. The key assumption is stateless communication for scalability and reliability. The server does not store any client state

[8] https://www.w3.org/People/Frystyk/thesis/WWW.html

between requests. Each request from a client contains all the necessary information to respond to the request. Any state about the interaction is stored in the client.

The *Hypertext Transfer Protocol* (HTTP) relies on the URI naming mechanism and provides a protocol for client/server communication. It is a very simple request/response protocol where the client sends a request message, and the server replies with a response message providing a way to publish and retrieve, e.g., HTML pages. Following the REST architecture, it is stateless.

An *HTTP request* consists of an HTTP request method (e.g., GET, PUT, POST, DELETE), the request URI, and HTTP protocol version information, optionally a list of HTTP headers consisting of name/value pairs, and optionally a message body. An *HTTP response* consists of an HTTP protocol version information and an HTTP status code, optionally a list of HTTP headers consisting of name/value pairs, and optionally a message body.

5.3 HTML and XML

The *Hypertext Markup Language (HTML)*[9] can be used to encode hypertext documents. In 1995, HTML 2.0 was specified as an IETF RFC 1866 (http://tools.ietf.org/html/rfc1866). The next version, HTML 3.2, reached a W3C recommendation status in 1997. HTML 5 was initiated in 2004, and ultimately, it was finalized within the W3C process (http://www.w3.org/TR/html5/). It facilitates a hypermedia environment. Documents use elements to "markup" or identify sections of content for different purposes or display characteristics. HTML markup consists of several types of entities, including elements, attributes, data types, and character references. Markup elements are not seen by the user when a page is displayed. The documents are rendered by browsers that interpret HTML. Hereby, a user agent (Web browser) uses links to enable users to navigate to other pages and to display additional information.

The *eXtensible Markup Language (XML)* is a language for creating other languages (dialects) and data exchange on the Web. For example, HTML can be expressed in XML as XHTML. It can be used for describing structured and semi-structured data. It is platform independent and has wide support providing interoperability. It is a W3C Recommendation (standard).

The structure of XML documents consists of elements, attributes, and content. It has one root element in a document. Characters and child elements form the content. An *XML element* has the following syntax: *<name>contents</name>* with *<name>* is called the opening tag, and *</name>* is called the closing tag. Element names are case sensitive.

[9] Adapted from https://en.wikipedia.org/wiki/HTML

Example. XML tags

```
<gender>Female</gender>
<story>Once upon a time there was.... </story>
```

XML can add attributes to XML elements. The syntax is *<name attribute_name = "attribute_value"> contents</name>*. Values are surrounded by single or double quotes.[10]

Example. XML attributes

```
<temperature unit="F">48</temperature>
<swearword language='fr'>con</swearword>
```

XML namespaces allow combined content from various XML sources. In general, documents use different vocabularies. Assume a scenario where the data from a store about products and an e-commerce system about product orders will be integrated. Merging multiple documents together can lead to name collisions. For example, the XML documents describing products and customers can both have *<name>* XML elements.

Namespaces provide a solution for name collision. Namespaces are a syntactic way to differentiate similar names in an XML document by bundling them using Uniform Resource Identifiers (URI), e.g., *http://example.com/NS*, which can be bound to a named or "default" prefix. The binding syntax uses the "xmlns" attribute to define a namespace.

- *Named prefix*: <a:foo xmlns:a = "http://example.com/NS"/>.
- *Default prefix:* <foo xmlns = "http://example.com/NS"/>.
- Element and attribute names are prefix – local part (or "local name") pairs, e.g., "http://example.com/NS", becomes "foo."

Example. XML namespaces

```
<?xml version='1.0' encoding='UTF-8'?>
 <order>
   <item code='BK3'>
     <name>FLATLAND: A Romance of Many Dimensions</name>
     <desc xmlns:html='http://www.w3.org/1999/xhtml'>
       The <html:b>best</html:b> book ever!
     </desc>
   </item>
 </order>
```

[10] Using XML attributes was the first way on the Web to include semantic information describing the content by the system Ontobroker (Fensel et al. 1998). It was called HTML-A as it was already applied to the HTML tag attribute Finally with RDFa this approach became a W3C standard, https://www.w3.org/MarkUp/2004/rdf-a

XML schema specifies to describe the elements in an XML document formally and is a W3C Recommendation. It can define a structure for XML documents and the names used by its XML schema language. Then an XML document can be verified according to these definitions. The main feature of an XML schema is the definition of types. An XML schema offers a set of built-in datatypes. They can be primitive datatypes (e.g., *xs:anyURI*, *xs:boolean*, *xs:date*, *xs:double*); simple types, which are typically derived from primitive datatypes; and complex types, which define the structure of elements. In general, a datatype can be derived from another datatype.

Example. A simple type called MyInteger that restricts the primitive integer type to the (1234, 5678] range

```
<xsd:simpleType name='MyInteger'>
  <xsd:restriction base='xsd:integer'>
    <xsd:minExclusive value='1234'/>
    <xsd:maxInclusive value='5678'/>
  </xsd:restriction>
</xsd:simpleType>
```

Example. A complex type of "person"

```
<xs:complexType name="person">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="age" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>
```

5.4 Is XML Schema an Ontology Language?

XML schema was developed as a means to specify a structure for documents (on the Web) (Klein et al. 2001). Therefore, it has a very specific *inheritance* relationship. A subdocument can refine but also extend the range of document properties. The subdocument is more specific. On one hand, it could provide less properties for describing the document, which is fine in terms of set-based inheritance. One the other hand, it may offer additional ways (ranges) to existing properties. This breaks any set-based interpretation of inheritance because an instance of a subdocument class is not a proper instance of its superdocument classes anymore. But it may make sense for documents. See the following example of inheritance as a means to extend a range definition:

Example. The broaderPerson type is a restriction of the person type. It uses a different range for the "age" element

```
<xs:complexType name="person">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="age" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="broaderPerson">
  <xs:complexContent>
    <xs:restriction base="person">
      <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
        <xs:element name="lastname" type="xs:string"/>
        <xs:element name="age" type="extendedAge"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:simpleType name="extendedAge">
  <xs:union memberTypes="xs:integer xs:string" />
</xs:simpleType>
```

The range of *age* in the subtype is broader than the range of its supertype. The *age* of *broaderPerson* allows *integer* and *string*; *person* allows *integer* only. This means the documents of type *broaderPerson* violate the definition of *person*, and an instance of type *broaderPerson* is not necessarily an instance of *person.* This is a violation of set-based inheritance, where a subset must be a real subset of the superset (see Fig. 5.4).

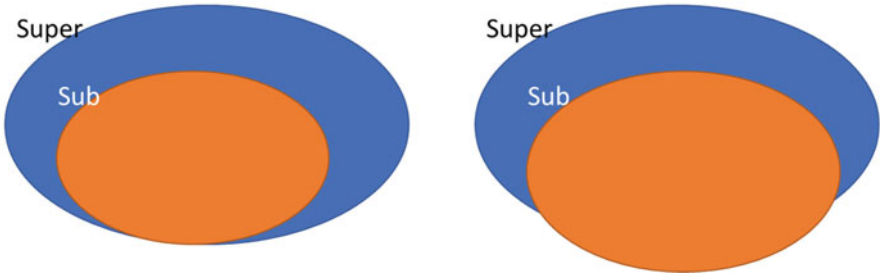


Fig. 5.4 Set-based inheritance (left) and its violation (right). XML violates the set-based interpretation of inheritance since the ranges of the properties on a subtype can be broader than the ones of the supertype

References

Fensel D, Decker S, Erdmann M, Studer R (1998) Ontobroker: the very high idea. In: Proceedings of the eleventh international florida artificial intelligence research society conference, May 18–20, 1998, AAAI Press, Sanibel Island, FL, pp 131–135

Fielding RT (2000) REST: architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California

Fielding RT, Taylor RN (2002) Principled design of the modern web architecture. ACM Transactions on Internet Technology (TOIT) 2(2):115–150

Klein M, Fensel D, Van Harmelen F, Horrocks I (2001) The relation between ontologies and XML schemas. Electronic Transactions on Artificial Intelligence (ETAI) 6(4)

Chapter 6
Natural Language Processing



Information retrieval (IR) deals with the retrieval of relevant information sources (typically documents) given a query. *Information extraction (IE)* (Pazienza 1997) goes one step further to obtain information from unstructured sources (e.g., documents). It is mostly concerned with processing unstructured text to "understand" the content and extract data following a certain structure. *Natural Language Processing (NLP)* is a field that addresses the challenge of processing, analyzing, and understanding natural language (Clark et al. 2012). Naturally, it plays an important role in information extraction, given the definition above. Actually, one of the first uses of the term "knowledge graph" was in the context of NLP. In the PhD thesis of Bakker (Bakker 1987), knowledge graphs were referred to as a way of structuring and representing knowledge extracted from a scientific text (see also the content about semantic nets in Chap. 2) (Maynard et al. 2016). Summing up, the major difference between information retrieval and NLP is that IR retrieves documents for a human to inspect, whereas NLP dives into the content of documents and tries to answer human queries directly.

An NLP application such as information extraction typically deals with three main tasks (Maynard et al. 2016)[1]:

- Linguistic processing
- Named entity recognition (NER), and
- Relation extraction

A pipeline of low-level linguistic tasks that prepare a given text for the next steps is provided in Fig. 6.1.

Tokenization It is the task of splitting an input text into atomic units called tokens. Tokens generally correspond to words, numbers, and symbols typically separated

[1] The explanations regarding the given NLP pipeline are mainly based on Maynard et al. (2016). See there for a more detailed description.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_6



Fig. 6.1 The linguistic processing pipeline (Figure adapted from Maynard et al. (2016))

with whitespace. Tokenization is typically the first step in any linguistic processing pipeline since more complex steps use tokens as input.

Sentence Splitting It is the task of separating a text into sentences. The main challenge is to decide if punctuation is at the end of a sentence or has another purpose. For instance, sentence splitters typically benefit from a list of abbreviations to decide if a full stop is at the end of a sentence or an abbreviation like Ms.

Part-of-Speech (POS) Tagging It is the task of labeling words with their linguistic categories, also known as part-of-speech (e.g., noun, verb). Several tag classifications exist, like Penn Treebank (PTB), Brown Corpus, and Lancaster-Oslo/Bergen Corpus.

Morphological Analysis and Stemming[2] Morphological analysis is the task of identifying and classifying the linguistic units of a word. For example, the verb "talked" consists of a root "talk" and a suffix "-ed." The stemming task strips a word of all its suffixes.

Parsing/Chunking It is the task of building the syntactic structure of sentences given a grammar (e.g., which verb connects to which nouns) and building a parse tree. Parsing shows how different parts of a sentence are related to each other. Parsers give very granular information about the words in a sentence; however, they may be computationally very expensive.

These preprocessing steps are needed to support higher-level tasks such as named entity recognition and relation extraction.

Named Entity Recognition Based on this linguistic processing, named entity recognition (NER) can be provided. NER is an annotation task helping to identify the semantics of people, organizations, locations, dates, and times mentioned in the text (e.g., Nelson Mandela, Amazon, New York . . .). The information obtained from the linguistic processing is valuable at this stage, e.g., nouns are good candidates for named entities. Ambiguity is a typical challenge, i.e., "Amazon" can refer to a rainforest or an organization. Gazetteer lookup is a step involving a simple lookup from a list of known entities. Again, gazetteer lookup can be ambiguous. London can be a location but also part of an organization. Combined with the result of POS tagging, the ambiguity may be solved relatively more easily. Some rule-based grammar matching can be combined with gazetteer lookup to improve effectiveness, e.g., a pattern might tell that "University of" is followed by a city. Finally, the coreference resolution aims to identify coreferences between named entities, e.g., the

[2] This step may look significantly different for different languages.

linking of pronouns: "I voted for Trump, because he was the best candidate," John said.

Relation Extraction It is typically considered a slot-filling task. Given a relation schema, what are the relations between named entities in the text? Such a relation schema can be based on an ontology. A relation extractor takes as input an annotated text with named entities, relationship occurrences, and linguistic features extracted by preprocessing for training, as well as testing instances for prediction. Its output is extracted relations.

As evident from the proposal of the Dartmouth AI workshop (McCarthy et al. 2006), one of the major parts of the AI vision was to enable computers to understand and generate natural language. This had also implications on information retrieval, particularly for answering (controlled) natural language queries based on unstructured text. SIR, a computer program for semantic information retrieval by Bertram Raphael,[3] was developed in 1964 (Raphael 1964). It was one of the first programs with "understanding" capabilities. It could learn word associations and property lists based on conversations in a controlled form of the English language. It could then answer questions based on the knowledge it gained from those conversations.

6.1 An Example: The GATE System

We introduce the *General Architecture for Text Engineering (GATE)*,[4] which is a Java suite of tools, as an example of an NLP tool set. It was originally developed at the University of Sheffield beginning in 1995 and is now used worldwide by a wide community of scientists, companies, teachers, and students. It can be used for many natural language processing tasks, including information extraction in many languages.[5] It became commercialized by Ontotext/Sirma via KIM.[6]

GATE Tokenizer The tokenizer splits the text into tokens such as numbers, punctuations, and different types of words; see Fig. 6.2.[7]

GATE Sentence Splitter The splitter annotates the sentences in a text. It uses a gazetteer list of abbreviations to determine whether a punctuation marks a sentence or has another purpose. In Fig. 6.3, the text is split into two sentences.

[3] https://en.wikipedia.org/wiki/Bertram_Raphael

[4] https://gate.ac.uk/

[5] https://en.wikipedia.org/wiki/General_Architecture_for_Text_Engineering, content distributed under CC-BY-SA 3.0

[6] The content is based on the documentation in https://gate.ac.uk/sale/tao/splitch6.html#x9-1190006

[7] The example used in this figure and all the upcoming figures is from https://www.bbc.com/news/world-europe-65215576

Evan Gershkovich, an experienced Russia reporter, was arrested last week in the city of Yekaterinburg while working for the Wall Street Journal (WSJ).

The WSJ released another statement following news of the charges: "As we've said from the beginning, these charges are categorically false and unjustified, and we continue to demand Evan's immediate release."

Type	Set	Start	End	Id	Features
Token		0	4	1...	{kind=word, length=4, orth=upperInitial, string=Evan}
Token		5	16	1...	{kind=word, length=11, orth=upperInitial, string=Gershkovich}
Token		16	17	1...	{kind=punctuation, length=1, string=,}
Token		18	20	1...	{kind=word, length=2, orth=lowercase, string=an}
Token		21	32	1...	{kind=word, length=11, orth=lowercase, string=experienced}
Token		33	39	1...	{kind=word, length=6, orth=upperInitial, string=Russia}
Token		40	48	1...	{kind=word, length=8, orth=lowercase, string=reporter}
Token		48	49	1...	{kind=punctuation, length=1, string=,}

Fig. 6.2 The GATE tokenizer

Evan Gershkovich, an experienced Russia reporter, was arrested last week in the city of Yekaterinburg while working for the Wall Street Journal (WSJ).

The WSJ released another statement following news of the charges: "As we've said from the beginning, these charges are categorically false and unjustified, and we continue to demand Evan's immediate release."

Type	Set	Start	End	Id	Features
Sentence		0	150	3...	{}
Sentence		152	359	3...	{}

Fig. 6.3 The GATE sentence splitter. Two sentences are marked on the bottom of the figure with their start and end characters

GATE Part-of-Speech Tagger (POS-Tagger) It annotates tokens in the tokenized text with a part-of-speech tag. Figure 6.4 shows the tagged tokens (annotated with the start- and end-character positions). The "category" feature specifies the POS-Tag.

GATE Gazetteer The gazetteer provides a list of entity names that are used to match named entities in the text. A gazetteer lookup can be seen as a preliminary named entity recognition. Each gazetteer list represents a set of names, such as names of cities, organizations, and so on. In Fig. 6.5, we use the default GATE gazetteer. The token *Evan* is tagged as a male person.

GATE Semantic Tagger The semantic tagger is a named entity recognition module that uses certain rules on previously annotated text to make more advanced named entity recognition in comparison to the gazetteer-based step. This module can

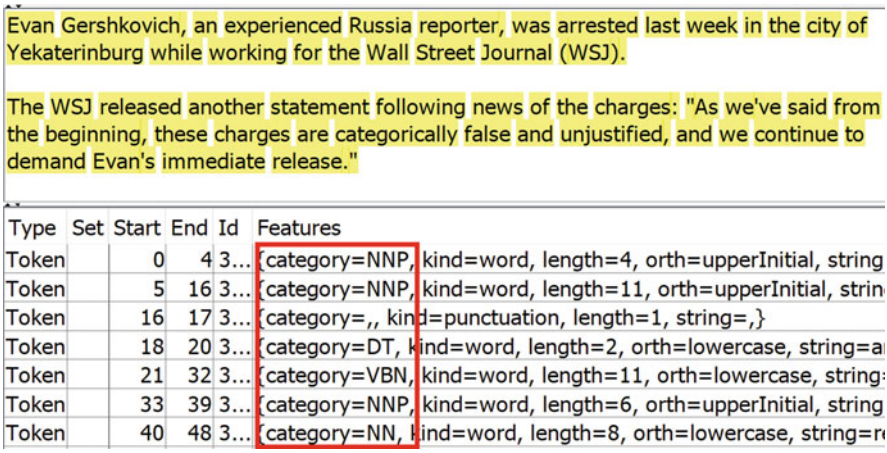


Fig. 6.4 The GATE POS-tagger

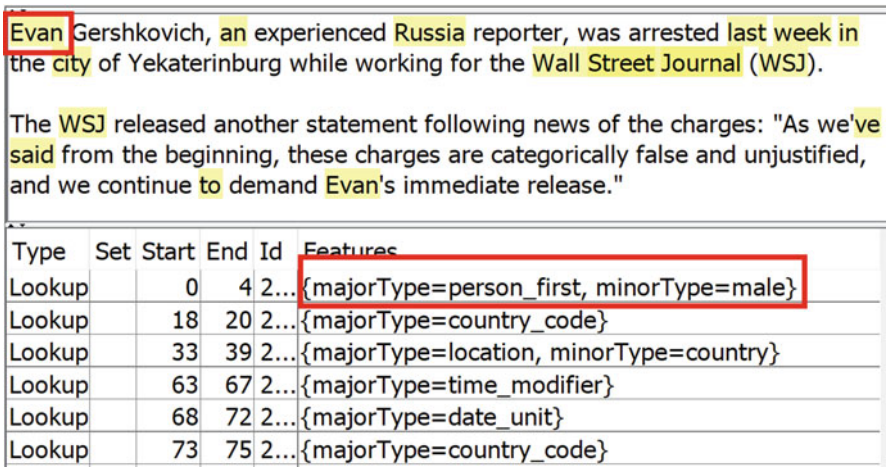
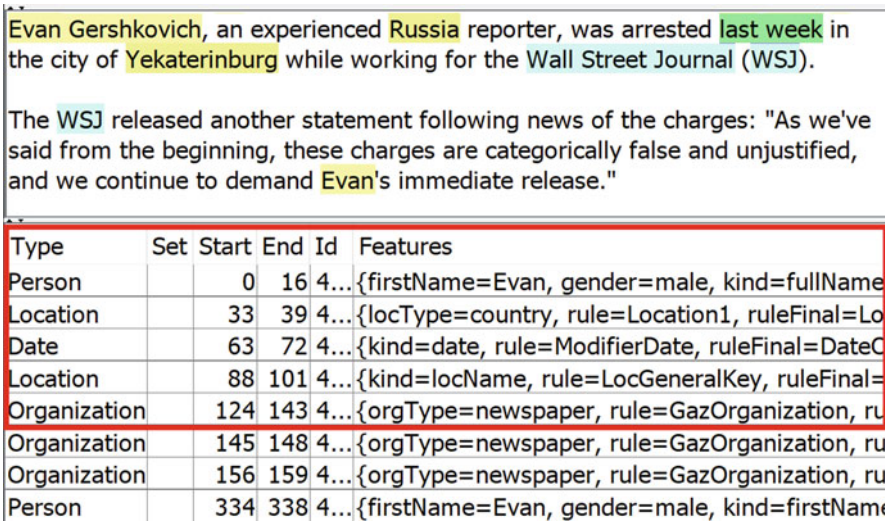


Fig. 6.5 The GATE gazetteer results

recognize entities like *people*, *organizations*, *locations*, *monetary amounts*, *dates*, *percentages*, and some types of *addresses* (see Fig. 6.6).

GATE – OrthoMatcher (Orthographic Coreference) This module creates identity relations between the recognized named entities to perform a coreference resolution (e.g., the mention of the same name at multiple places in the text or the usage of aliases and synonyms). Figure 6.7 shows that the OrthoMatcher matches the occurrences of "Wall Street Journal" and "WSJ."

GATE—Pronominal Coreference Resolution This module also does coreference resolutions like the OrthoMatcher, but it links named entities and the pronouns

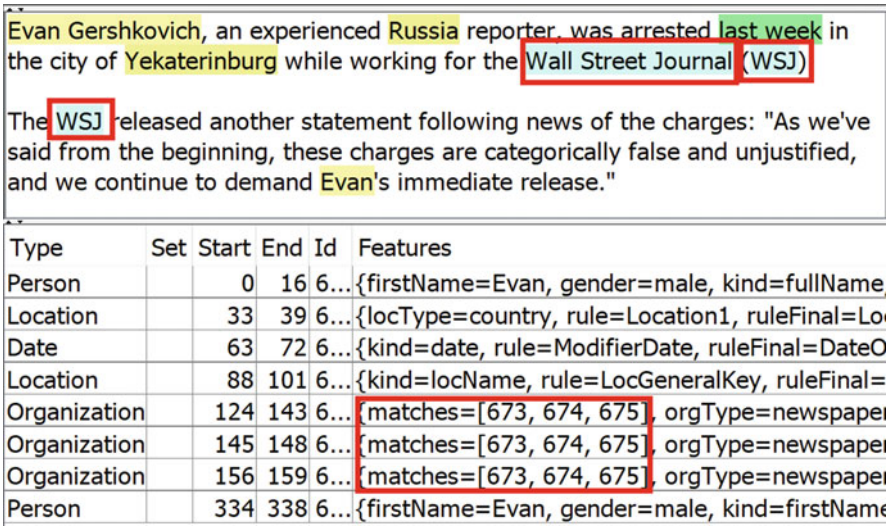


Evan Gershkovich, an experienced Russia reporter, was arrested last week in the city of Yekaterinburg while working for the Wall Street Journal (WSJ).

The WSJ released another statement following news of the charges: "As we've said from the beginning, these charges are categorically false and unjustified, and we continue to demand Evan's immediate release."

Type	Set	Start	End	Id	Features
Person		0	16	4...	{firstName=Evan, gender=male, kind=fullName
Location		33	39	4...	{locType=country, rule=Location1, ruleFinal=Lo
Date		63	72	4...	{kind=date, rule=ModifierDate, ruleFinal=DateC
Location		88	101	4...	{kind=locName, rule=LocGeneralKey, ruleFinal=
Organization		124	143	4...	{orgType=newspaper, rule=GazOrganization, ru
Organization		145	148	4...	{orgType=newspaper, rule=GazOrganization, ru
Organization		156	159	4...	{orgType=newspaper, rule=GazOrganization, ru
Person		334	338	4...	{firstName=Evan, gender=male, kind=firstNam

Fig. 6.6 The GATE—NER module recognizing people and organizations



Evan Gershkovich, an experienced Russia reporter, was arrested last week in the city of Yekaterinburg while working for the Wall Street Journal (WSJ).

The WSJ released another statement following news of the charges: "As we've said from the beginning, these charges are categorically false and unjustified, and we continue to demand Evan's immediate release."

Type	Set	Start	End	Id	Features
Person		0	16	6...	{firstName=Evan, gender=male, kind=fullName
Location		33	39	6...	{locType=country, rule=Location1, ruleFinal=Lo
Date		63	72	6...	{kind=date, rule=ModifierDate, ruleFinal=DateO
Location		88	101	6...	{kind=locName, rule=LocGeneralKey, ruleFinal=
Organization		124	143	6...	{matches=[673, 674, 675], orgType=newspaper
Organization		145	148	6...	{matches=[673, 674, 675], orgType=newspaper
Organization		156	159	6...	{matches=[673, 674, 675], orgType=newspaper
Person		334	338	6...	{firstName=Evan, gender=male, kind=firstNam

Fig. 6.7 The GATE— OrthoMatcher module recognizing coreferences

referring to them. Figure 6.8 shows an excerpt from the results of this module. The pronoun *his* is matched with the named entity *Evan Gershkovich.*

Almost all GATE components can be configured extensively with new rules and grammar. A plug-in system gives flexibility for NLP pipeline development. For example, the OntoGazetteer plugin can be used in the NER task to match entities with classes in an ontology.

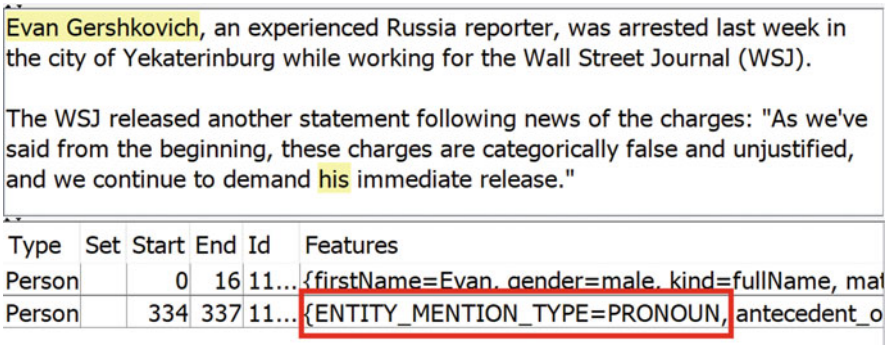


Fig. 6.8 The GATE—pronominal coreference resolution module recognizing pronouns

References

Bakker R (1987) Knowledge graphs: representation and structuring of scientific knowledge (doctoral dissertation). University Twente

Clark A, Fox C, Lappin S (eds) (2012) The handbook of computational linguistics and natural language processing, vol 118. Wiley

Maynard D, Bontcheva K, Augenstein I (2016) Natural language processing for the semantic web. In: Ding Y, Groth P (eds) Synthesis lectures on the semantic web: Theory and technology, vol 15. Morgan & Claypool Publishers, pp 1–184

McCarthy J, Minsky ML, Rochester N, Shannon CE (2006) A proposal for the dart-mouth summer research project on artificial intelligence, August 31, 1955. AI Magazine 27(4):12–12

Pazienza MT (ed) (1997) Information extraction: A multidisciplinary approach to an emerging information technology, vol LNAI 1299. Springer

Raphael B (1964) SIR: a computer program for semantic information retrieval. Doctoral Dissertation, MIT

Chapter 7
Semantic Web: Or AI Revisited



When we left AI, we were talking about the so-called AI winter. Building knowledge-based systems was too costly to justify their construction and usage besides in some niche applications. However, the world did not stop rotating. With the Web, a new, extremely large information source appeared. And the content was provided for free by a worldwide crowd activity. It just required turning this information into machine-understandable knowledge by adding semantics. Then the so-called knowledge acquisition bottleneck would be a memory from the far past. And precisely, this happened with the *Semantic Web*. The Semantic Web started in 1996 for two reasons. First was the dramatic growth of the World Wide Web.

The semantic annotations of content were used to improve search by applying machine-understandable semantics.

Figure 7.1 shows an early initiative (Fensel et al. 1997; Benjamins et al. 1999) and system (Fensel et al. 1998) using the semantic annotation of Web content based on an ontology to support information retrieval and extraction, i.e., direct query answering.

The Web was designed to bring a piece of information to a human user. The user has to manually extract and interpret the provided information and may need to integrate it with information from other sources. This can turn into a huge human effort, especially when the information comes from different Web sites and needs to be carefully and properly aligned. For example, I want to travel from Innsbruck to Rome, where I want to stay in a hotel and visit the city.

Many Web sites have to be visited, manually checked, aligned, and backtracked when one aspect provides a bottleneck. By adding semantics, a virtual agent can do all these tasks on behalf of the human user. It can also automatically provide links to additional information, e.g., this image is about Innsbruck, Dieter Fensel is a professor, etc. Syntactic structures are converted to knowledge structures (Fig. 7.2).

Obviously, one could argue that platforms such as Booking.com provide such a service for traveling based on backend integration, too. However, then you leave the

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_7

Table 1. Some Ontology Definitions

Concept Hierarchy	Attribute Definitions	Rules

Object[].
Person :: Object.
Employee :: Person.
AcademicStaff :: Employee.
Researcher :: AcademicStaff.
Publication :: Object.
	Person[
	firstName =>> STRING;
	lastName =>> STRING;
	eMail =>> STRING;
	…
	Publication =>> Publication].
		FORALL Person1, Publication1
		Publication1 : Publication
		[autor ->> Person1]
		<->
		Person1:Person
		[publication ->> Publication1].

```
<html> <body>
        <a onto=„page:Researcher“>
        <h2>Welcome on my homepage</h2>
        My name is <a onto=[firstName=body]“> Richard</a>
         <a onto=[LastName=body]“> Benjamins</a>.
</body></html>                                    HTML-A 1996
```

Fig. 7.1 The knowledge acquisition initiative



Fig. 7.2 Semantic enrichment of content

open Web and deal with a large provider through a single Web site, i.e., you are locked into a walled garden.[1]

The second reason to work on the Semantic Web (mostly for researchers) was to solve the knowledge acquisition bottleneck. The vision of the Semantic Web was to build a *brain of/for humankind* (Fensel and Musen 2001). Billions of humans put information on this global network. Through this, the Web mirrors large fractions of human knowledge. Empowered by semantics, computers can access and understand this knowledge. The knowledge acquisition problem would be solved when the entire humanity would join this task for free. Like annotating content with structural info such as HTML, it just requires annotating content with semantic information.

The second half of the 1990s witnessed the initial efforts to allow the Web to scale by enabling machines to consume it (see Fensel et al. 2005):

[1] Tim Berners-Lee Warns of ‘Walled Gardens’ for Mobile Internet, https://archive.nytimes.com/www.nytimes.com/idg/IDG_002570DE00740E1800257394004818F5.html

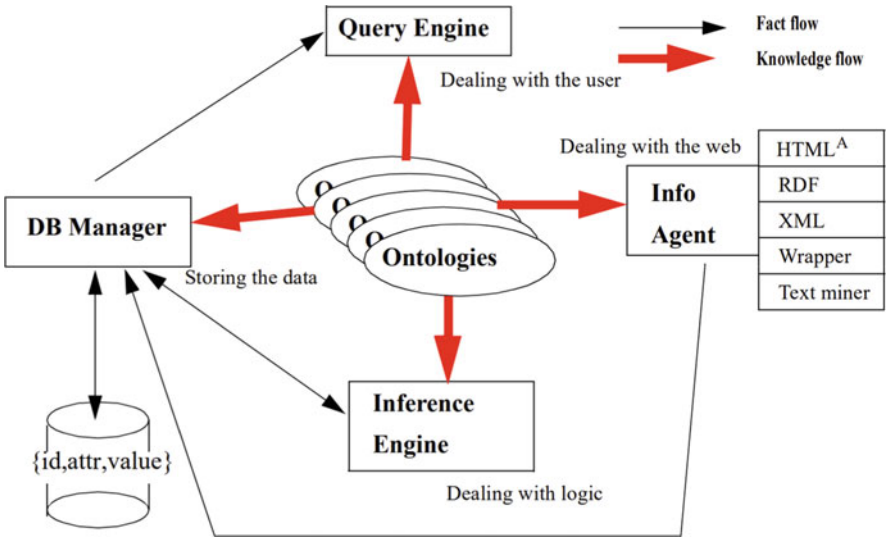


Fig. 7.3 The architecture of On2broker from a bird's-eye view (Fensel et al. 1999)

- SHOE (Heflin et al. 2003) is an early system for semantically annotating Web content in a distributed manner.
- Ontobroker and On2Broker (Fensel et al. 1998, 1999) are an architecture for consuming distributed and heterogeneous semi-structured sources on the Web (Fig. 7.3).

With ontologies at its center, *On2Broker* consists of four main components: a query interface, a repository, an info agent, and an inference engine. The *Query Engine* provides a structured input interface that enables users to define their queries. Input queries are then transformed into the query language (e.g., SPARQL). The *DB Manager* decouples query answering, information retrieval, and reasoning and provides support for the materialization of inferred knowledge.

The *Info Agent* extracts knowledge from different distributed and heterogeneous data sources. HTML-A[2] pages and Resource Description Framework (RDF) repositories can be included directly. HTML and XML data sources require processing provided by wrappers to derive RDF data. The *Inference Engine* relies on knowledge imported from the crawlers and axioms contained in the repository to support advanced query answering (Fensel et al. 1999).

These early academic systems triggered a significant standardization effort by the W3C to develop common standards for building such Semantic Web systems. The *Semantic Web Stack* (Fig. 7.4) has been developed that contains a set of specifications that enables the Semantic Web based on existing Web recommendations. Core

[2] A predecessor of RDFa.

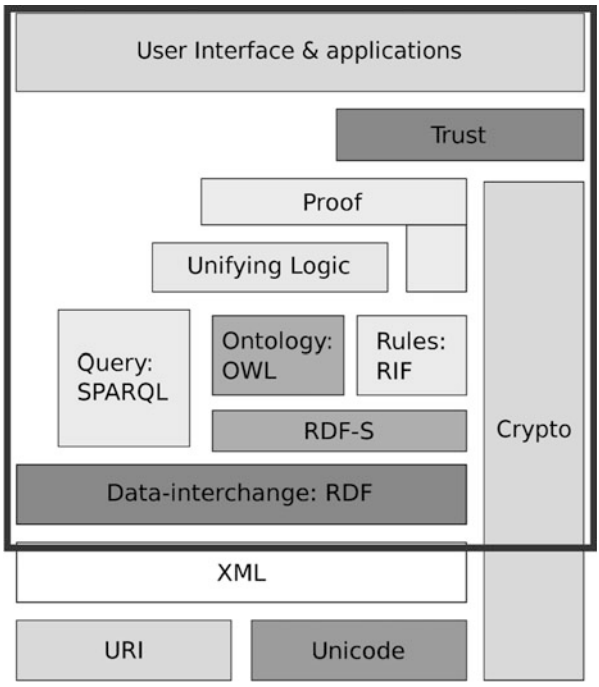


Fig. 7.4 The W3C Semantic Web layer cake (Image by Mhermans, distributed under CC-BY-SA license https://commons.wikimedia.org/wiki/File:SW_layercake_2006.svg)

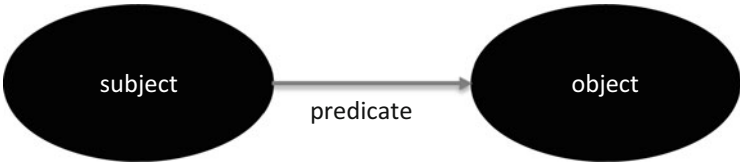


Fig. 7.5 The RDF data model

technologies are RDF as a data model, SPARQL as a query language, and RDFS and OWL as ontology languages.

The *Resource Description Framework (RDF)* provides a triple-based data model for exchanging (meta)data on the Web. Resources can be identified with Internationalized Resource Identifiers (IRIs). Shared IRIs provide the means for linking resources and forming a directed labeled graph (Fig. 7.5).

RDF Schema (RDFS) and the *Web Ontology Language (OWL)*[3] are ontology languages for defining the meaning of RDF data. RDFS provides the means for defining type and property taxonomies and domains/ranges for properties. OWL extends RDFS with universal and existential quantifiers; inverse, functional, and

[3] Meanwhile further developed to OWL2. https://www.w3.org/TR/owl2-primer/

```
SELECT ?event ?name
WHERE {
  ?event :name ?name .
  ?event :location ?location .
  ?location :city :Innsbruck .
}
```

Fig. 7.6 An excerpt from a simple SPARQL query that returns the event instance identifiers and their names for events in Innsbruck

transitive properties and cardinality restrictions; and more using description logic as underlying logical formalism.

SPARQL is a query language for manipulating RDF(S) data. It resembles the Structured Query Language (SQL), which is used for relational databases. It mainly works by matching graph patterns to the triples in a graph (Fig. 7.6).

Still, it took longer than expected before these techniques took over the Web at full scale. Google was excellent at finding information based on simple syntactic means, and semantic annotations did not seem to be necessary at all. However, this changed drastically when *search engines* (finding links people follow and immediately leaving the search page) tried to turn into *query-answering engines* that want to engage with their users. Here, formal understanding and, therefore, semantics are a must. Suddenly, large search engines turned from strong opponents into enthusiastic supporters of the Semantic Web.

Schema.org was started in 2011 by Bing, Google, Yahoo!, and Yandex to provide vocabularies for annotating Web sites. Meanwhile, it has become a de facto standard for annotating content and data on the Web with around 797 types, 1457 properties, 14 datatypes, 86 enumerations, and 462 enumeration members (in March 2023). It provides a corpus of types (e.g., LocalBusiness, SkiResort, Restaurant) organized hierarchically), properties (e.g., name, description, address), range restrictions (e.g., Text, URL, PostalAddress), and enumeration values (e.g., DayOfWeek, EventStatusType, ItemAvailability) and covers a large number of different domains.

The use of semantic annotations has experienced a tremendous surge in activity since the introduction of schema.org. According to WebDataCommons, 50% of all crawled Web pages contain annotations (Fig. 7.7).

Crawl Date	October 2022	
Total Data	82.71 Terabyte	(compressed)
Parsed HTML URLs	3,048,746,652	
URLs with Triples	1,518,609,988	
Domains in Crawl	33,820,102	
Domains with Triples	14,235,035	
Typed Entities	19,072,628,514	
Triples	86,462,816,435	
Size of Extracted Data	1.6 Terabyte	(compressed)

Fig. 7.7 WebDataCommons statistics from October 2022 (http://webdatacommons.org/structureddata/2022-12/stats/stats.html)

References

Benjamins VR, Fensel D, Decker S, Perez AG (1999) KA2: building ontologies for the internet: a mid-term report. International Journal of Human-Computer Studies 51(3):687–712

Fensel D, Musen MA (2001) The semantic web: a brain for humankind. IEEE Intell Syst 16(2): 24–25

Fensel D, Erdmann M, Studer R (1997) Ontology groups: semantically enriched subnets of the www. In: Proceedings of the 1st international workshop intelligent information integration during the 21st German annual conference on artificial intelligence, Freiburg, September

Fensel D, Decker S, Erdmann M, Studer R (1998) Ontobroker: the very high idea. In: Proceedings of the eleventh international Florida artificial intelligence research society conference, May 18–20, 1998, Sanibel Island, FL, AAAI Press, pp 131–135

Fensel D, Angele J, Decker S, Erdmann M, Schnurr H, Staab S, Studer R, Witt A (1999) On2broker: Semantic-based access to information sources at the WWW. In: Proceedings of the IJCAI-99 workshop on intelligent information integration, held on July 31, 1999, in conjunction with the sixteenth international joint conference on Artificial Intelligence City Conference Center, Stockholm, CEUR-WS.org, CEUR Workshop Proceedings, vol 23. https://ceur-ws.org/Vol-23/fensel-ijcai99-iii.pdf

Fensel D, Hendler JA, Lieberman H (eds) (2005) Spinning the Semantic Web: bringing the World Wide Web to its full potential, Paperback edition. MIT Press

Heflin J, Hendler JA, Luke S (2003) SHOE: a blueprint for the semantic web, pp 29–63. In: Fensel et al. (2005)

Chapter 8
Databases



While the AI community was being built in the 1960s, there have been important developments on the data side of things, too.[1] NASA needed a system to keep track of parts and suppliers for rocket parts in the Apollo project. So they developed a hierarchical file system for it. Then IBM built a database based on this file system, called the Information Management System (IMS),[2] in 1966. Hierarchical models store data in a tree-like form. Around the same time, Charles Bachman from General Electric came up with the network model and implemented it in the Integrated Data Store (IDS) (Bachman 2009), which stores data in a graph-like form.

Both the *hierarchical model* and *network model* had some crucial characteristics. Both databases were navigational. Programs were accessing database tuple at a time. This means writing many nested loops. In consequence, the performance of the query is in the hands of the programmer. For example, IMS even did not have any abstraction of the storage implementation. You need to write different loops for different data structures.

This coupling opened the possibility for accessing data very efficiently but also meant that a slight change in the database structure required the applications to be reprogrammed. A mathematician at IBM named Edgar F. Codd saw this situation and wanted to achieve *data independence*. In 1970, Codd produced a model that decouples a logical representation of a database from its implementation (physical vs logical). He created the *relational model* for databases (Codd 1990). Data are stored in simple data structures (relations), and operations on the database are not at the tuple level but at the relation level. This allows for working on many tuples at once. Operations are formalized with relational algebra, which is based on set operations and can be queried with a high-level declarative language. The query language SQL (aka SEQUEL) has become an American National Standards Institute (ANSI)

[1] The content of this chapter is partially based on (Pavlo 2020).

[2] https://www.ibm.com/docs/en/zos-basic-skills?topic=now-history-ims-beginnings-nasa

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_8

Customer (Customer ID, Tax ID, Name, Address, City, State, Zip, Phone, Email, Sex)

Customer ID	Tax ID	Name	Address	[More fields...]
1234567890	555-5512222	Ramesh	323 Southern Avenue	...
2223344556	555-5523232	Adam	1200 Main Street	...
3334445563	555-5533323	Shweta	871 Rani JhansiRoad	...
4232342432	555-5325523	Sarfaraz	123 Maulana Azad Sarani	...

Fig. 8.1 The relational model (Adapted from https://en.wikipedia.org/wiki/Relational_model)

standard, and the relational model dominated the field instead of navigational models like hierarchical and network-based models.

The relational model stores data in tuples in a structure called relation (table). A tuple is a partial function that maps attribute names to values, and a relation consists of a header (a finite set of attribute names (columns)) and a body (a set of tuples that all have a header as their domain; see Fig. 8.1).[3]

However, there were always alternative developments in this area. Meanwhile, object-oriented programming was becoming popular, and developers realized an issue between the object-oriented paradigm and the relational model. Actually, there is a relation-object mismatch: complex objects are not straightforwardly stored in a relational database, e.g., a customer with multiple phone numbers can be represented with a customer object with an array of phone numbers. In relational databases, this would ideally require two relations. For object-oriented databases, queries can be written natively with the programming language of the application, most of the time at the expense of declarative querying. *Object-oriented databases*[4] were able to store more complex data models with the help of features like the native definition of custom types and inheritance.

Deductive databases (Ullman and Zaniolo 1990; Ramakrishnan and Ullman 1995) tried to provide logic as an access layer for databases quite in parallel to the developments of knowledge representations on the AI side. As a result of the efforts in that direction, the *Datalog* language was created (Ceri et al. 1989).

It is aligned with the relational model formalism but is still more expressive. Later, SQL implemented some features of Datalog-like recursion. Datalog is a subset

[3] Based on https://en.wikipedia.org/wiki/Relational_model, licensed under CC-BY-SA

[4] https://en.wikipedia.org/wiki/Object_database

of Prolog: expressivity traded off for computational efficiency. Pure Datalog has no function symbols and no negation. Without function symbols and negation, a Datalog program always guarantees to terminate. Also, it is fully declarative, i.e., the order of statements does not matter as it does for Prolog. Finally, it has efficient query optimization methods like Magic sets (Bancilhon et al. 1985).

The strong connection between object-oriented databases and procedural languages was hindering data independence. *F-logic* (Kifer and Lausen 1989) emerged as a way to build deductive object-oriented databases combining object orientation with logical languages. F-logic is a declarative language for creating, accessing, and manipulating object-oriented database (OODB) schemas and provides a higher-order knowledge representation formalism that combines features of frames and object-oriented languages. F-logic supports overriding in inheritance and "statements about properties"[5] (in F-logic, they are referred to as attributes); properties are defined locally on classes and have closed-world and unique name assumptions. Although it was initially targeting databases, it also has many use cases in AI, especially in frame-based systems. F-logic and description logic (DL) represent two camps in knowledge representation (de Bruijn et al. 2005). We have already covered that description logic uses the open-world assumption and does not have a unique name assumption.

- F-logic adopts the *closed-world assumption*: the train timetable example does not require an explicit statement of nonexisting connection as the facts that do not exist are considered false.
- F-logic adopts the *unique name assumption*: remember the example from description logic – if the same individual is married to more than one thing, DL says these things are the same (unless they are explicitly stated as different). F-logic says there is an inconsistency.

The relational model has certain limitations for working with knowledge graphs. Relational databases have rigid schemas, which are good for data quality and optimization (query and storage) but work poorly for integrating heterogeneous and dynamic sources. Let us view some approaches on how to host a knowledge graph with a relation database (Ben Mahria et al. 2021):

- The most straightforward approach is using a *statement table*. The graph is stored in one table with three columns (subject, predicate, object).
- The *property-class table* provides one table for each type.
- *Vertical partitioning* provides one table per property.
- Finally, one can provide *virtual RDF graphs* over a relational database. It is a relatively popular way to convert relational databases to knowledge graphs because it allows the integration of the knowledge graph into already-existing IT environments.

[5] Without moving into Second Order Logic semantically.

However, each of these approaches comes with various issues, which we will discuss in more detail in Chap. 19 on knowledge graph hosting.

With rapidly growing and dynamic data, traditional relational model solutions reached similar limitations, e.g., for geospatial data, graphical data, the Internet of Things (IoT), social networks, etc. Big tech companies like Amazon, eBay, Facebook, and Google developed their own ad hoc solutions to scale up. This showed that for many modern cases, answering user queries fast is more important than ensuring immediate consistency/integrity. *NoSQL* solutions have appeared (e.g., document stores, key-value databases, graph databases). The typical features of such databases are not having rigid schemas, nonrelational models, and mostly custom application programming interfaces (APIs) for data access and manipulation.

In the context of knowledge graphs, one important NoSQL data model is the *graph model* (Angles and Gutierrez 2008). Various graph data models have been around for a long time, but as a commercial success, two models seem to have won the race:

- RDF graph databases that support directed labeled edge models (e.g., GraphDB)
- Property graphs that take edges as first-class citizens (e.g., Neo4j)

RDF databases (aka triple stores) have been around longer as a result of the Semantic Web effort and are much more standardized than property graphs. For example, GraphDB[6] is an enterprise-level graph database with RDF and SPARQL support. Like many other RDF graph databases, it has three main functionalities: storage, querying, and reasoning. It supports Named Graphs (Carroll et al. 2005) and RDF-Star (Hartig and Champin 2021) for reification and various indexing mechanisms, including adapters for external indexing systems like Lucene and Elasticsearch. A strength is its customizable and modular reasoning support. It supports various rule sets for reasoning with different expressivity and complexity:

- Standard RDFS.
- RDFS+: RDFS extended with symmetric and transitive properties.
- Various OWL dialects: OWL-Lite, OWL-QL, OWL-RL, and OWL Horst (ter Horst 2005).
- Customized rule sets for reasoning can be defined.
- Finally, constraint checking with SHACL is provided.

Summarizing the discussion, there are two main options. The first is using a virtual graph on top of a well-established relational database. This brings the advantage that the handling of the knowledge graph is directly integrated into the existing standard infrastructure. However, access and manipulation require mappings to an ontology. Such mappings can be complex, and many things can go wrong. Also, the work with the graph is limited as no built-in reasoning is provided. The second option is to use an RDF repository. They are directly customized for the

[6] https://graphdb.ontotext.com/

data model and semantics of a knowledge graph and provide interchangeability by relying solely on open W3C standards.

References

Angles R, Gutierrez C (2008) Survey of graph database models. ACM Computing Surveys (CSUR) 40(1):1–39

Bachman CW (2009) The origin of the integrated data store (IDS): the first direct-access DBMS. IEEE Ann Hist Comput 31(4):42–54

Bancilhon F, Maier D, Sagiv Y, Ullman JD (1985) Magic sets and other strange ways to implement logic programs. In: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, Cambridge, MA, March 24–26, pp 1–15

Ben Mahria B, Chaker I, Zahi A (2021) An empirical study on the evaluation of the RDF storage systems. Journal of Big Data 8:1–20

Carroll JJ, Bizer C, Hayes P, Stickler P (2005) Named graphs. J Web Semantics 3(4):247–267

Ceri S, Gottlob G, Tanca L et al (1989) What you always wanted to know about Datalog (and never dared to ask). IEEE Trans Knowl Data Eng 1(1):146–166

Codd EF (1990) The relational model for database management: version 2. Addison-Wesley Longman Publishing Co., Inc.

De Bruijn J, Lara R, Polleres A, Fensel D (2005) Owl dl vs owl flight: conceptual modeling and reasoning for the semantic web. In: Proceedings of the 14th international conference on World Wide Web, Chiba, May 10–14, pp 623–632

Hartig O, Champin PA (2021) Metadata for RDF statements: the RDF-Star approach. Lotico

Kifer M, Lausen G (1989) F-logic: a higher-order language for reasoning about objects, inheritance and schema. In: SIGMOD/PODS04: international conference on management of data and symposium on principles database and systems, Portland, OR, June 1, pp 134–146

Pavlo A (2020) 01 - History of databases (CMU databases/Spring 2020). https://www.youtube.com/watch?v=SdW5RKUboKc

Ramakrishnan R, Ullman JD (1995) A survey of deductive database systems. J Log Program 23(2): 125–149

ter Horst HJ (2005) Combining RDF and part of OWL with rules: semantics, decidability, complexity. In: The Semantic Web–ISWC 2005: 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6–10, 2005. Proceedings 4, Springer, pp 668–684

Ullman JD, Zaniolo C (1990) Deductive databases: achievements and future directions. ACM SIGMOD Rec 19(4):75–82

Chapter 9
Web of Data



We will introduce the Linked Data concept and its extension to Linked Open Data.

9.1 Linked Data

Starting around 2006, *Linked Data* is a set of principles to publish interlinked data on the Web, less focused on the complex formalisms to describe the data. It is mostly an adaptation of the traditional Web principles extending them from content to data. How do we extend this Web of documents into a Web of Data? Typically, the data are published in an isolated fashion, for example, embedded into Web pages or Resource Description Framework (RDF) datasets. Assume that one data silo contains movies, another one contains reviews, and, again, another one contains actors. Many common things are represented in multiple datasets. Linking identifiers (i.e., Uniform Resource Identifiers (URIs)) would link these datasets (Bizer et al. 2008).

The Web of Data is envisioned as a global database consisting of objects and their descriptions; objects are linked with each other and have a high degree of object structure with explicit semantics for links and content. Finally, it should be designed for humans and machines (Bizer et al. 2008) and was defined by Tim Berners-Lee in 2006 with the aim of providing a unified method for describing and accessing resources (Käfer 2020):

> The Semantic Web isn't just about putting data on the Web. It is about making links, so that a person or machine can explore the Web of Data. With Linked Data, when you have some of it, you can find other, related data.—Sir Tim Berners-Lee

Recall the traveling example used in Chap. 7 to illustrate the need for automatic data integration. Data integration involves combining data residing in different sources and providing users with a unified view of these data. Data integration over the Web can be implemented as follows (Herman 2012):

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_9

- Export the datasets to be integrated as RDF graphs.
- Merge identical resources (i.e., resources having the same URI) from different datasets.
- Start making queries on the integrated data, queries that were not possible on the individual datasets.

Linked Data should follow certain principles:[1,2]

1. Uniform Resource Identifiers (URIs) should be used to name and identify individual things.
2. Hypertext Transfer Protocol (HTTP) URIs should be used to allow these things to be looked up, interpreted, and subsequently "dereferenced."[3]
3. Useful information about what a name identifies should be provided through open standards such as RDF.
4. When publishing data on the Web, other things should be referred to using their HTTP URI-based names.

Principle 1: Use URIs as names for things This principle implies the use of URIs not only for documents but also for any resource inside RDF graphs. This allows for addressing a unique resource while exchanging data. For example, *https://www.wikidata.org/wiki/Q1735* is a URI for the city of Innsbruck in Austria, and so is https://dbpedia.org/resource/Innsbruck.

Principle 2: Use HTTP URIs so that users can look up those names The users can be both humans and automated agents. The server delivers a suitable representation of the requested resource via HTTP with the help of content negotiation (Fig. 9.1). A resource can be anything, an image, a document, a person... An HTTP URI (more precisely a URL) both identifies and locates a resource and also specifies how to access it.

Principle 3: When someone looks up a URI, provide useful information using the following standards RDF is the standard data model for both Semantic Web and Linked Data. The provided information via a URI should be in RDF. There are

[1] http://www.w3.org/DesignIssues/LinkedData.html See also Linked Data Platform 1.0. W3C Recommendation 1.0.

[2] See also a nice tutorial about Linked Data and Knowledge Graphs in Käfer (2020).

[3] Note that this principle implies that URLs must be used for identifying resources. The main purpose of URLs is to "locate" resources and determine how they should be accessed (i.e., via a protocol like HTTP, FTP...). Using URLs for both identification and access may be problematic, as the way we access things may change over time. A good example of such a scenario is the shift from HTTP to HTTPS, which may mean that many published resources on the Semantic Web become inaccessible. The "same" URI with HTTP and HTTPs can locate the same resource but are not inherently treated as the same identifier. See also a discussion about this in the Semantic Web mailing list: https://lists.w3.org/Archives/Public/semantic-web/2023Jun/0028.html

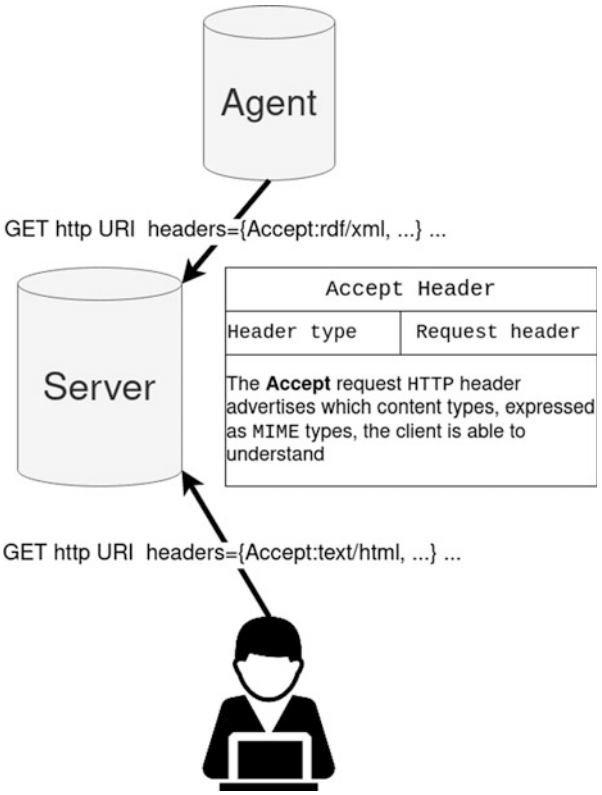


Fig. 9.1 Linked data principle 2 (Human icon by freepik. https://www.flaticon.com/free-icon/computer_44357)

various serializations for RDF, such as RDF/XML, Turtle, Notation 3, and JSON LD.

Principle 4: Include links to other URIs Resources across datasets must be linked. In principle, we can talk about three different kinds of links (Heath and Bizer 2011):

- *Relationship links*: links to external entities related to the original entity, e.g., *<https://elias.kaerle.com> schema:knows <https://umutcanserles.com>*
- *Identity links*: links to external entities referring to the same object or concept, e.g., *dbpedia:Innsbruck owl:sameAs wikidata:Innsbruck, e.g., dbpedia:Innsbruck skos:exactMatch http://globalwordnet.org/ili/i83317*
- *Vocabulary links*: links to definitions of the original entity, e.g., *dbo:City owl:equivalentClass schema:City*

In principle, any property could be used to create links across datasets; there are some properties commonly used for various linking purposes:

- *owl:sameAs*—for connecting individuals
- *owl:equivalentClass*—for connecting classes
- *rdfs:seeAlso*—provides additional information about resources

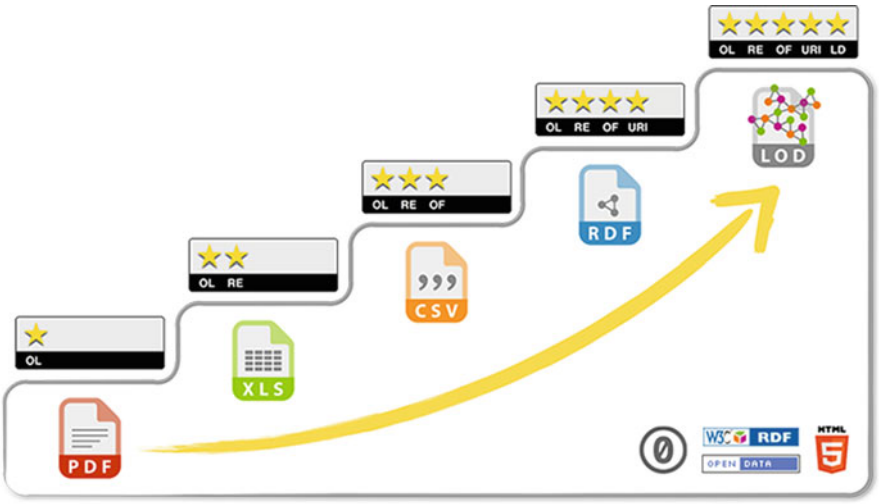


Fig. 9.2 The 5-star linked open data rating scheme (https://5stardata.info)

- *skos:closeMatch*, *skos:narrowMatch*, *skos:broaderMatch*, *skos:relatedMatch*, and *skos:exactMatch*—for linking "concepts" across datasets with weak formal semantics.[4]

Linked Open Data is an extension to Linked Data principles, which will be discussed next.

9.2 Linked Open Data

Linked Data is not necessarily open. The definition of Linked Data has been extended, through the openness principle by Tim Berners Lee, thereby coining the term Linked Open Data in 2010. It aims to encourage institutions (especially governments) to provide "good" Linked Data:

> Linked *Open* Data (LOD) is Linked Data which is released under an open licence, which does not impede its reuse for free.—Sir Tim Berners-Lee[5]

Linked Open Data essentially implies that the information must be published with an open license. An example of an open license is Creative Commons CC-BY. For calling them Linked Open Data, a few more aspects should be considered. These aspects are provided on a scale of 5 stars (Fig. 9.2):[6]

[4] See Sect. 13.4 about Simple Knowledge Organization System (SKOS).
[5] https://www.w3.org/DesignIssues/LinkedData.html
[6] https://5stardata.info/

- 1-star: Make your data available on the Web in some format under an open license.
- 2-star: Make it available as structured data (e.g., Excel instead of PDF).
- 3-star: Make it available in a nonproprietary open format (e.g., CSV instead of Excel).
- 4-star: Use URIs to denote things so that people can point at your data.
- 5-star: Link your data to other data to provide context.

In the following, we will discuss a possible process for publishing Linked Open Data.

LOD: A Linked Data Publication Scenario in Seven Steps (Heath and Bizer 2011)

1. Select vocabularies. Important hereby is the reuse of existing vocabularies to increase the value of your dataset and align your own vocabularies to increase interoperability.
2. Partition the RDF graph into "data pages" (for example, a data page can contain the RDF representation of a specific person).
3. Assign a URI to each data page.
4. Create Hypertext Markup Language (HTML) variants of each data page (to allow the rendering of pages in browsers). It is important to set up content negotiation between RDF and HTML versions.
5. Assign a URI to each entity (cf. "Cool URIs for the Semantic Web").
6. Add page metadata. It is important to make data pages understandable for consumers, i.e., add metadata such as publisher, license, topics, etc.
7. Add a semantic sitemap.

Data need to be prepared (e.g., extracted from text), links and the proper usage of URIs need to be defined, and these data need to get stored in an appropriate storage system and published as data on the Web (see Fig. 9.3 for an architecture for publishing Linked Data on the Web) (Heath and Bizer 2011). For example, the RDF graph in Fig. 9.4 contains information about the book "The Glass Palace" by Amitav Ghosh.[7] Information about the same book, but in French this time, is modeled in the RDF graph in Fig. 9.5. We can merge identical resources (i.e., resources having the same URI) from different datasets (Figs. 9.6 and 9.7).

Finally, one can make queries on the integrated data. A user of the second dataset may ask queries like "give me the title of the original book." This information is not in the second dataset. However, this information can be retrieved from the integrated dataset, in which the second dataset is connected with the first dataset (Herman 2012). For example, DBpedia Mobile (Becker and Bizer 2009) combines maps on mobile devices with information about places from DBpedia, pictures from Flickr, reviews from Revyu, etc.[8]

[7] Figures 9.4, 9.5, 9.6, and 9.7 are taken from W3C (Herman 2012)—last accessed on 04.04.2023. The content is distributed under CC-BY 3.0.

[8] Herman (2012) http://wiki.dbpedia.org/DBpediaMobile

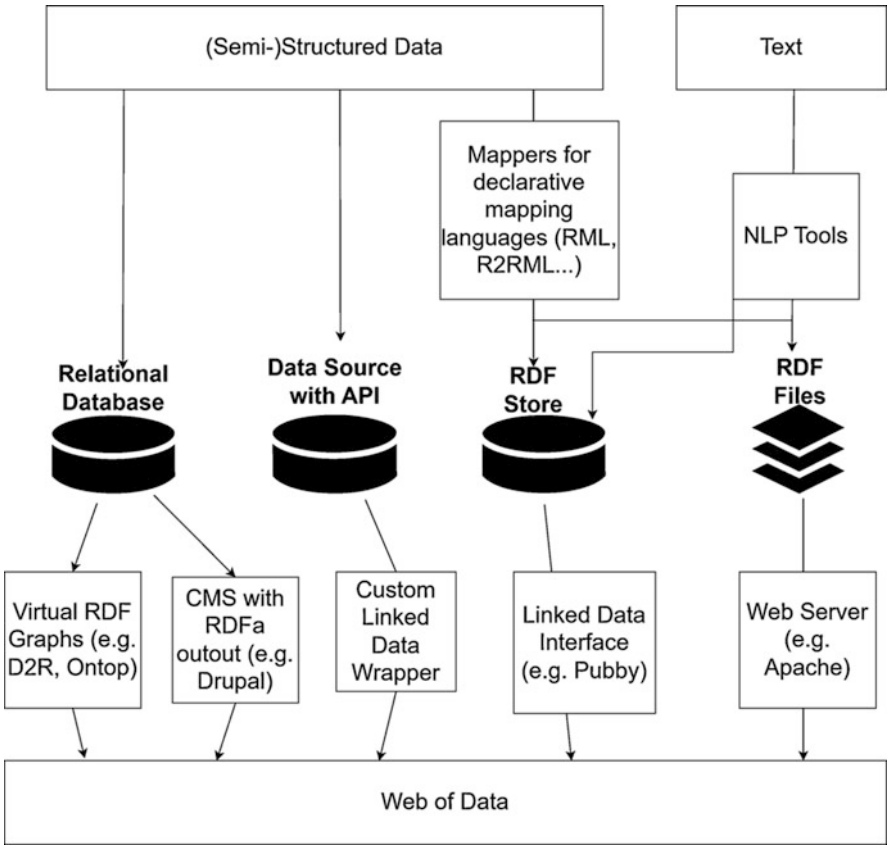


Fig. 9.3 An architecture for publishing linked open data (Figure adapted from Heath and Bizer (2011))

Starting from the early years of development, special browsers, data mashups, and search engines have been developed for consuming Linked Data (Heath and Bizer 2011):

- *Linked Data browsers*: to explore things and datasets and to navigate between them, several browsers have been developed. For example, the Tabulator Browser (MIT, USA) (Berners-Lee et al. 2006), Marbles (FU Berlin, DE),[9] and OpenLink RDF Browser[10] (OpenLink, UK) have been developed.
- *Linked Data mashups*: sites that mash up (thus combine) Linked Data were developed, for example, Revyu.com (KMI, UK), DBpedia Mobile (Becker and

[9] https://mes.github.io/marbles/

[10] https://www.w3.org/wiki/OpenLinkDataExplorer

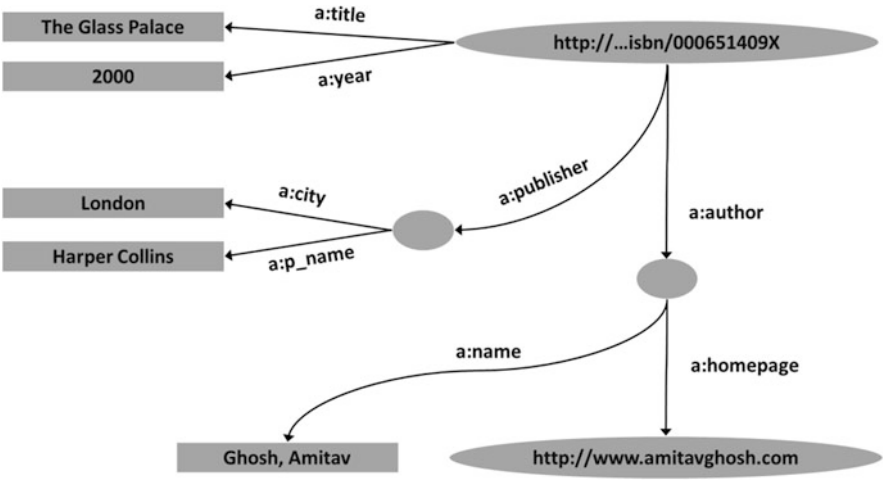


Fig. 9.4 The RDF graph contains information about the book "The Glass Palace" by Amitav Ghosh

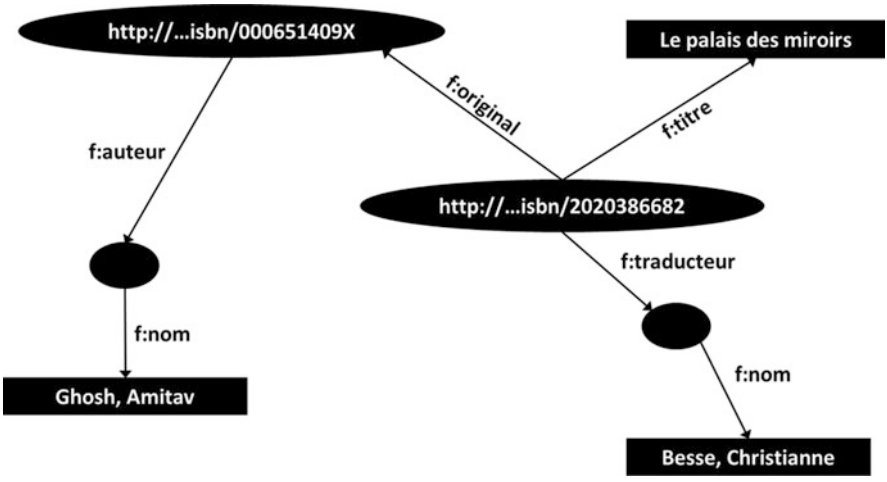


Fig. 9.5 The same book is modeled in a French dataset

Bizer 2009) (FU Berlin, DE), and Semantic Web Pipes (Le-Phuoc et al. 2009) (DERI, Ireland).

- *Search engines for searching Linked Data*: examples are Falcons (Cheng and Qu 2009) (IWS, China), Sindice (Tummarello et al. 2007) (DERI, Ireland), and Swoogle (Ding et al. 2004) (UMBC, USA).

In May 2020, the LOD cloud contained 1300 datasets containing subclouds in the following categories (Fig. 9.8):

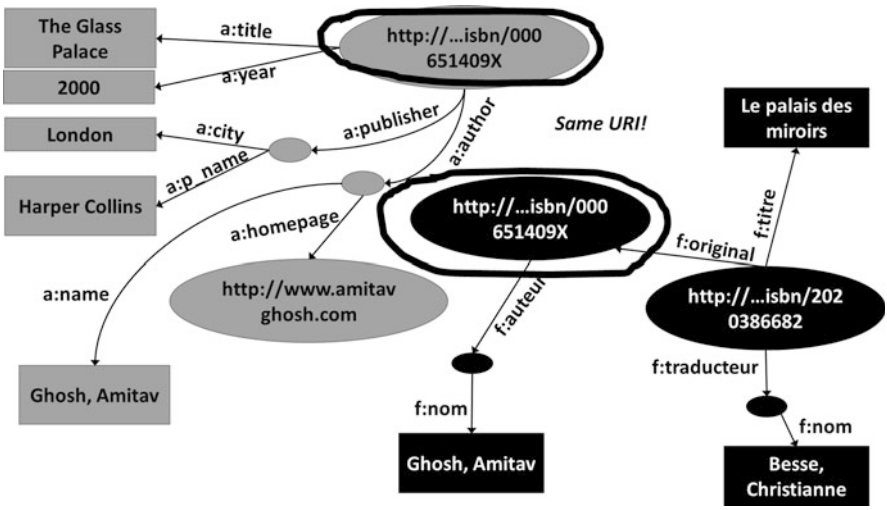


Fig. 9.6 Two datasets with identical resources

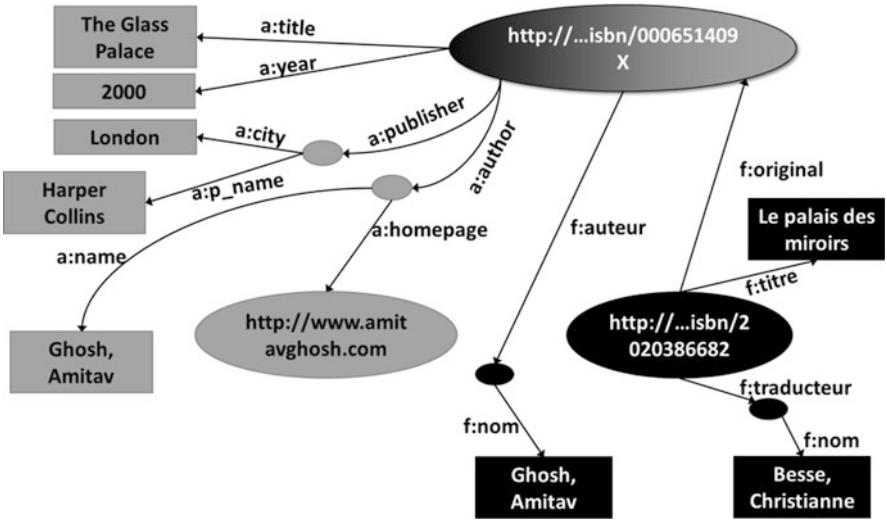


Fig. 9.7 Merging identical resources from Figs. 9.4 and 9.5

- Cross-domain: e.g., DBpedia, WebIsALOD, Linked Open Colors
- Geography: weather stations, Smart Points of Interest
- Government: UNESCO statistics, Italian immigration statistics
- Life Science: several BioPortal and Bio2RDF datasets
- Linguistics: WordNet, Ontologies of Linguistic Annotation
- Others

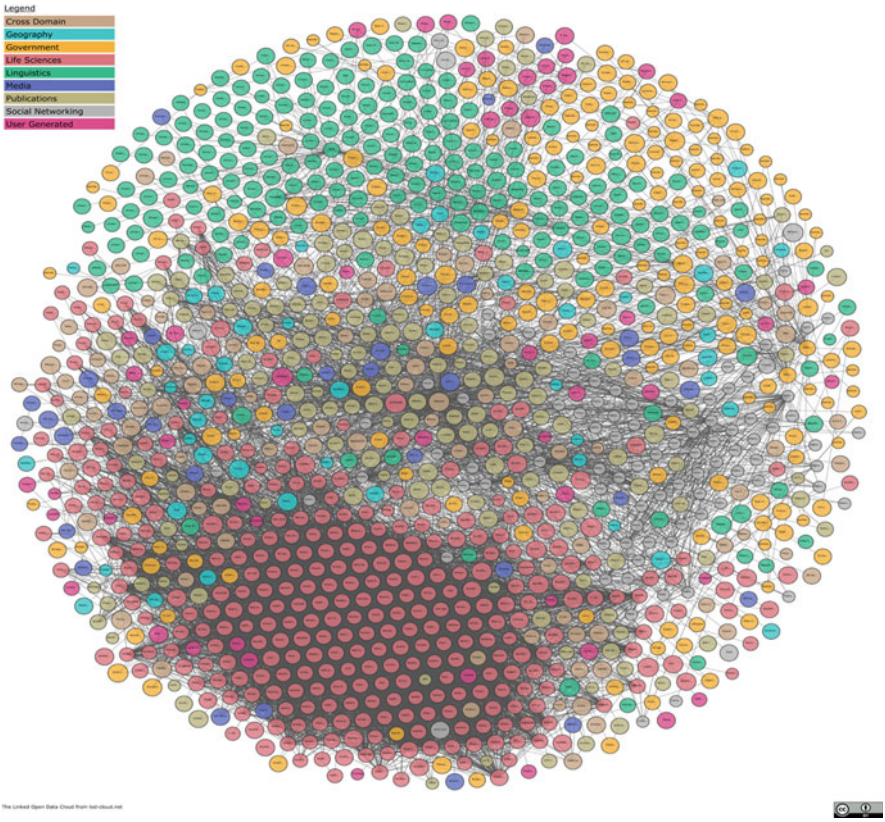


Fig. 9.8 The linked open data cloud from May 2020 (Image by https://lod-cloud.net/, distributed under CC-BY)

These interconnected datasets of Linked Data *are the predecessor and enabler for knowledge graphs.*

References

Becker C, Bizer C (2009) Exploring the geospatial semantic web with DBpedia mobile. Journal of Web Semantics 7(4):278–286

Berners-Lee T, Chen Y, Chilton L, Connolly D, Dhanaraj R, Hollenbach J, Lerer A, Sheets D (2006) Tabulator: exploring and analyzing linked data on the semantic web. In: Proceedings of the 3rd international semantic web user interaction workshop, Athens, November 6, vol 2006, p 159

Bizer C, Heath T, Berners-Lee T (2008) Linked data: principles and State of the Art. 17th International World Wide Web Conference W3C Track @ WWW2008, Beijing, China, April 23–24

Cheng G, Qu Y (2009) Searching linked objects with falcons: approach, implementation and evaluation. International Journal on Semantic Web and Information Systems (IJSWIS) 5(3): 49–70

Ding L, Finin T, Joshi A, Pan R, Cost RS, Peng Y, Reddivari P, Doshi V, Sachs J (2004) Swoogle: a search and metadata engine for the semantic web. In: Proceedings of the 13th ACM international conference on information and knowledge management, Washington, DC, November 8–13, pp 652–659

Heath T, Bizer C (2011) Linked data: evolving the web into a global data space. Synthesis lectures on the semantic web: theory and technology, vol 1(1), pp 1–136

Herman I (2012) SW tutorial. http://www.w3.org/People/Ivan/CorePresentations/SWTutorial

Käfer T (2020) Distributed knowledge graphs: knowledge graphs and linked data, AI4Industry summer school. https://ai4industry.sciencesconf.org/data/DistributedKnowledgeGraphsPt.1.pdf

Le-Phuoc D, Polleres A, Hauswirth M, Tummarello G, Morbidoni C (2009) Rapid prototyping of semantic mashups through semantic web pipes. In: Proceedings of the 18th international conference on World Wide Web, Madrid, April 20–24, pp 581–590

Tummarello G, Delbru R, Oren E (2007) Sindice. com: Weaving the open linked data. In: The Semantic Web: 6th international semantic web conference, 2nd Asian semantic web conference, ISWC 2007+ ASWC 2007, Busan, November 11–15, 2007. Proceedings, Springer, pp 552–565

Chapter 10
Knowledge Graphs



AI research started with great ambitions and big visions; however, these illusions have disappeared after the realization of the knowledge acquisition bottleneck. It was actually the Semantic Web and its application Linked Data that solved this problem by basically crowdsourcing the acquisition of knowledge. Finally, schema.org has helped solve this problem with the right incentives and a simple data model motivating most Web site owners to empower the sites with semantics. Databases are still dominated by the relational model, but NoSQL databases, including graph databases, have shown their use and gained increasing adoption in recent years for working with knowledge graphs. A good example of how these lanes of research and development converge to knowledge graphs is Google's business:

- Google builds a knowledge graph, mostly based on schema.org annotations embedded on Web pages.
- It uses schema.org as an ontology for the knowledge graph and stores it in a graph database.
- It answers queries are directly based on the knowledge graph, instead of only retrieving documents based on statistical analysis.

We will *partially define* this exciting knowledge graph technology that was influenced by years of research in various areas.

What is a knowledge graph? A knowledge graph is a large semantic net that integrates heterogeneous data sources into a graph structure. Instead of providing a large set of rules for deducing answers to questions, it contains many explicit facts, which is its main difference from traditional knowledge-based systems. Finally, a graph is a mathematical structure in which some pairs in a set of objects are somehow related.[1] The graph in Fig. 10.1 is an example of a knowledge graph about events, represented as a directed edge-labeled graph. Nodes represent entities, and edges represent the relationship between them. A core feature of knowledge

[1] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_10

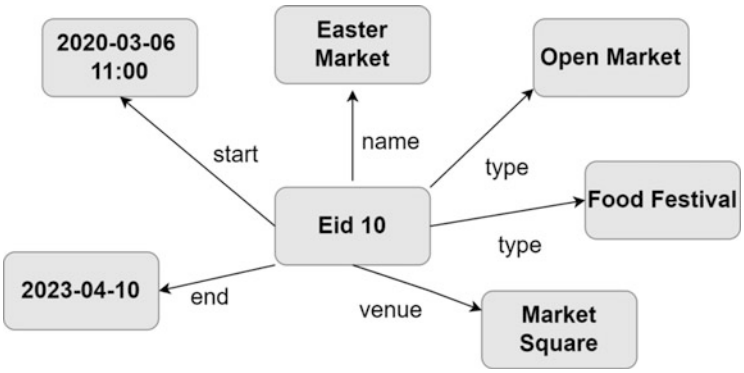


Fig. 10.1 A knowledge graph excerpt, adapted from Hogan et al. (2021)

Table 10.1 Numerical overview of some knowledge graphs (Paulheim 2017)

Name	Instances	Facts	Types	Relations
DBpedia (English)	4,806,150	176,043,129	735	2813
YAGO	4,595,906	25,946,870	488,469	77
Freebase	49,947,845	3,041,722,635	26,507	37,781
Wikidata	15,602,060	65,993,797	23,157	1673
NELL	2,006,896	432,845	285	425
OpenCyc	118,499	2,413,894	45,153	18,526
Google's Knowledge Graph	570,000,000	18,000,000,000	1500	35,000
Google's Knowledge Vault	45,000,000	271,000,000	1100	4469
Yahoo! Knowledge Graph	3,443,743	1,391,054,990	250	800

graphs is that there is no strict distinction between schema and data: everything is just nodes and edges. This makes knowledge graphs very flexible when it comes to integrating heterogenous sources. No need to obey or update a rigid schema like in relational databases. This also makes them relatively easier to grow in size. Still, some edges and nodes may come from a controlled vocabulary and have a special meaning, which describes the meaning of the data (i.e., knowledge graphs may use an ontology).

Knowledge graphs are supposed to be large scale, i.e., they should be able to handle billions to trillions of data. Table 10.1 enumerates some of them.

So, *what* knowledge graphs are:

- A big mess of *frillions* of facts describing the world from multiple points of view.
- A Rosetta stone allows humans and machines to exchange meaning.
- Merging smart and big data into explainable artificial intelligence.
- Capturing a large fraction of human knowledge and explicitly turning it into a brain for/of humankind.
- A thrilling area of research and engineering with a large impact and deep issues to be resolved.

Why (or for what) are knowledge graphs important? Knowledge graphs are enabling technology for:

- Turning search into query answering (the original purpose for developing them) and starting to enter dialogs and interactions with human users (chatbots)
- Virtual agents (information search, eMarketing, and eCommerce)
- Cyber-physical systems (the Internet of Things, smart meters, etc.)
- Physical agents (drones, cars, satellites, androids, etc.)

Because knowledge is power, you look foolish without it. The statistical analysis or matrix multiplication of large data volumes can bring you a long way but lacks the integration of world knowledge and understandable interaction (*explainable AI*) with humans. Turning knowledge graphs into a useful resource for problem-solving is not a trivial task as there are various challenges, such as size/scalability, heterogeneity, dynamic and active data, as well as quality, i.e., correctness and completeness. This book will provide you with a conceptual framework, a formal underpinning based on logic, and a methodology for building knowledge graphs based on these two pillars.

References

Hogan A, Blomqvist E, Cochez M, d'Amato C, Melo GD, Gutierrez C, Kirrane S, Gayo JEL, Navigli R, Neumaier S et al (2021) Knowledge graphs. ACM Comput Surv 54(4):1–37

Paulheim H (2017) Knowledge graph refinement: a survey of approaches and evaluation methods. Semant Web 8(3):489–508

Part II
Knowledge Representation

Chapter 11
Introduction to Knowledge Representation



Semantics is "the study of reference, meaning, or truth."[1] Different fields, like philosophy, linguistics, mathematics, and computer science study semantics from different perspectives. In Part II of the book, we talk about what we mean by semantics and what it entails in the scope of knowledge representation and knowledge graphs.

In different scenarios, we can talk about different kinds of semantics. Lexical semantics studies the meaning of terms based on their relationship to other terms. Intensional semantics deals with meaning based on the properties an instance should have. For example, natural numbers between 1 and 10, not including 1 and 10, can be represented with the following set notation:

$$A = \{x \in N \mid 1 < x < 10\}$$

Extensional semantics gives the meaning of a concept based on all its elements. The same set above can be defined by enumerating its elements:

$$A = \{2, 3, 4, 5, 6, 7, 8, 9\}$$

Prototype semantics gives the meaning based on a prototypical instance. For example, specific kinds of animals can be defined by adding and removing different features to a prototypical animal of that kind. Finally, the pragmatical semantics theory claims that meaning is given to concepts and objects only by their use and interaction by the actors.[2]

In the introduction part, we defined knowledge graphs as "very large semantic nets that integrate heterogeneous data sources in certain domains." But what is semantic about a knowledge graph? There is nothing inherently semantic about a

[1] https://en.wikipedia.org/wiki/Semantics

[2] https://en.wikipedia.org/wiki/George_Herbert_Mead

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_11

graph. It is just labeled nodes and edges. When we talk about semantics for knowledge graphs, we mean two things:

- Computer understanding: formal semantics defined by mathematical means.
- Human understanding: the symbols are mapped to entities and relationships in a domain.

First, we will examine the semantics for knowledge graphs and the relationship between human and computer understanding within the framework of the five levels of knowledge representation from Brachman (Brachman 1979) in Chap. 12. In Chaps. 13 and 14, we go deeper into two of these levels, namely epistemology and logic, respectively. For epistemology, we take different languages for knowledge graphs and introduce their modeling primitives. For the logic chapter, we cover the underlying logical formalisms used by those knowledge representation languages. Due to its importance on the Web, we take an analysis of the schema.org[3] ontology as an example of the conceptual level and how it makes use of the three underlying layers, as well as what it provides for the natural language layer above, in Chap. 15. Chapter 16 will provide a brief summary of Part II.

Reference

Brachman RJ (1979) On the epistemological status of semantic networks. In: Associative networks. Elsevier, pp 3–50

[3] Schema.org

Chapter 12
The Five Levels of Representing Knowledge



Brachman defines five levels of knowledge representation (Brachman 1979). Each level corresponds to an explicit set of primitives at the service of a knowledge graph developer. These levels are:

1. Implementation level
2. Logical level
3. Epistemological level
4. Conceptual level, and
5. Linguistic level

The *implementation level* deals with how knowledge is encoded by a computer. Same systems can be implemented with different programming languages. Remember the knowledge-based systems we have seen in Chap. 2. As an example, for knowledge-based systems at the implementation level, we can give a MYCIN expert system, which was developed originally with the LISP programming language, and production rules were encoded in the syntax of this specific language (Fig. 12.1) (Van Melle 1978). The same system is later also implemented with the Ruby language (Fig. 12.2).[1]

Knowledge-based systems like MYCIN can be also implemented with deductive logical rules instead of production rules. Prolog, for instance, is a popular language for expert systems. In fact, E-MYCIN was developed based on the original MYCIN implementation with production rules as an expert system shell in the Prolog language (Ulug 1986). An expert system shell is exactly what it sounds like – it is a "shell" that is stripped down from the domain-specific aspects of an expert system framework that allows it to be adopted for different domains. Figure 12.3 shows a MYCIN-like rule in Prolog.[2] The head is a (sub)goal in the system, which aims to decide whether an organism is a bacteroid with a certain confidence value, while

[1] Figure adapted from http://lazax.com/software/Mycin/mycin.html

[2] Adapted to the E-MYCIN rule structure.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_12

```
PREMISE: ($AND (SAME CNTXT GRAM GRAMNEG)
               (SAME CNTXT MORPH ROD)
               (SAME CNTXT AIR ANAEROBIC))
ACTION: (CONCLUDE CNTXT IDENTITY BACTEROIDES TALLY .6)

If 1) The gram stain of the organism is gramneg, and
   2)The morphology of the organism is rod, and
   3) The aerobicity of the organism is anaerobic

Then: There is a suggestive evidence (0.6) that the
identity of the organism is bacteroides
```

Fig. 12.1 Rule 035 of MYCIN encoded in LISP (Van Melle 1978)

```
(defrule 35
    if  (gram   organism        is  neg   )
        (morphl organism        is  rod   )
        (air   organism         is  anaerobic)
    then 0.6
        (identity   organism    is  bacteroides)
)
```

Fig. 12.2 Rule 035 of MYCIN encoded in RUBY

```
organism(CNTXT,N,'bacteroides' ,0.6) :-
eval_premise(same, gram, CNTXT ,N, 'neg' ,true),
eval_premise(same, morphl, CNTXT ,N, 'rod', true),
eval_premise(same, air, CNTXT ,N, 'anaerobic', true),
conclude(CNTXT, N, organism, 'bacteroides' ,0.6, 1 .0) .
```

Fig. 12.3 The implementation of Rule 035 from MYCIN with Prolog

eval_premise in the body builds the premise. The *conclude* function adds the inferred statement in the knowledge base as a fact.

As for knowledge graphs, implementation typically happens with the serialization syntaxes of modeling languages, for example, Resource Description Framework Schema (RDF(S)) and Web Ontology Language (OWL). The implementation level does not only contain languages but also engines that make use of these languages to work with knowledge. Rule-based systems typically use a rule engine or a reasoner. For example, for the storage and querying of knowledge graphs, triplestores and graph databases are used, while to reason with knowledge, there are many reasoner implementations for languages, like OWL.

At the *logical level*, knowledge is represented with logical primitives. These primitives can be predicates, functions, operators, propositions, and quantifiers,

depending on the logical formalism used. Knowledge representation at the logical level usually has model-theoretic semantics, i.e., these symbols are mapped on a mathematical structure with known properties. Decidable subsets of first-order logic (description logic) and minimal model semantics (rule languages) are applied at the logical level. These topics are introduced in Chap. 14.

Historically, knowledge representation on computers was happening at the logical and implementation level. Newell called these two levels the symbol level. In contrast, humans organize the knowledge they obtain from the world as concepts, instances, and their relationships. Newell classified conceptual and linguistic levels as the *knowledge level* (Newell 1981). Brachman argued that there is another level that links the representation at the conceptual level (i.e., humans understanding of the world) to the logical level (i.e., formal semantics encoded by symbols in the logical formalism), namely, the *epistemological level*. The epistemological level provides primitives such as concept types and structural relations. For example, concepts that refer to classes of objects that share certain characteristics (i.e., certain sets of properties and restrictions on those properties), attributes (i.e., relationships) with restrictions, inheritance relationships, instantiation relationships, instances, and values (i.e., relationship between instances). Among these primitives, relationships like inheritance and instantiation help structure the conceptual types in a knowledge representation. For knowledge graphs, languages like RDFS and OWL provide such primitives to model knowledge.

The *conceptual level* uses the primitives defined at the epistemological level to describe the entities in a domain. The primitives at this level are not dependent on a certain language. At the conceptual level, we are looking for answers to questions like, "What makes a concept of person," "What properties describe a person," and "How does a person relate to other concepts?" The modeling of knowledge graphs (in fact, typically any knowledge-based system) starts at this level. The most prominent way of modeling knowledge at the conceptual level is via building ontologies (see Sects. 2.5 and 18.1).

Unlike the conceptual level, knowledge representation at the *linguistic level* depends on a specific natural language. The primitives are words and linguistic expressions. The knowledge at this level is associated directly with the nouns and verbs of this specific language. The knowledge at this level is typically processed with Natural Language Processing (NLP) techniques.

Table 12.1 summarizes the five levels and the primitives they offer. In the remainder of this part, we will cover the epistemological and logical levels in more detail. The conceptual/ontological layer (see Sects. 2.5 and 18.1) and the linguistic level (see Chap. 6 and Sect. 18.2) are already covered in different parts of this book.

However, we will briefly analyze schema.org[3] in the framework of these five levels in Chap. 15 as an illustration as it is currently the most widely used ontology on the Web.

[3] Schema.org

Table 12.1 The five levels of knowledge representation and the modeling primitives they offer

Level	Language	Primitives
Linguistic level	Natural language	The primitives provided by a natural language, such as words, sentences, punctuation
Conceptual level	An ontology	Concepts and their relationships
Epistemological level	An ontology language	Definition of concept and slot types, structuring relationships like inheritance
Logical level	A logical formalism	Logical predicates, operators, quantifiers
Implementation level	A serialization format	Symbols of the language

References

Brachman RJ (1979) On the epistemological status of semantic networks. In: Associative networks. Elsevier, pp 3–50

Newell A (1981) The knowledge level: presidential address. AI Mag 2(2):1–1

Ulug F (1986) EMYCIN - prolog expert system shell. Technical report, Naval Postgraduate School, Monterey, CA

Van Melle W (1978) MYCIN: a knowledge-based consultation program for infectious disease diagnosis. International Journal of Man-Machine Studies 10(3):313–322

Chapter 13
Epistemology



The modelling of knowledge graphs typically starts at the conceptual level; however, the result of modelling by humans must be formalized for computers to understand it. The *epistemological level* provides modelling primitives that enable conceptual models to be encoded so the computers can understand them with the help of the underlying logical formalism. For modelling and manipulating knowledge graphs, several approaches have been developed in the past. In this chapter, we introduce some of those languages. The chapter is organized into three major aspects:

- In Sect. 13.1, we introduce the *data model* for Semantic Web-based knowledge graphs, namely, the Resource Description Framework (RDF)[1] and its schema language RDF Schema (RDFS),[2] as well as several tools that help with working with these languages.
- In Sect. 13.2, we introduce the approaches used for *data retrieval and manipulation* hosted by a knowledge graph. For querying, the SPARQL language[3] is described, and for applying constraints over data (and query results of SPARQL), the Shapes Constraint Language (SHACL)[4] is introduced.
- In Sect. 13.3, we cover the languages used for describing the data in a knowledge graph semantically, i.e., that provide *reasoning support over this data*. We introduce the Web Ontology Language (OWL)[5] and parts of the Rule Interchange Format (RIF).[6]

[1] https://www.w3.org.RDF/. The prefix *rdf* is used for the RDF namespace.

[2] https://www.w3.org/TR/rdf-schema/. The prefix *rdfs* is used for the RDFS namespace.

[3] https://www.w3.org/TR/rdf-sqarql-query/

[4] https://www.w3.org/TR/shacl/. The prefix *sh* is used for the SHACL namespace.

[5] https://www.w3.org/TR/owl2-overview/. The prefix *owl* is used for the OWL2 namespace.

[6] https://www.w3.org/TR/rif-overview/

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_13

We introduce the Simple Knowledge Organization System (SKOS)[7] in Sect. 13.4, a model for representing various knowledge organization systems (KOS) in RDF. SKOS has two major objectives:

- Providing a framework for ontology alignment and ontology networks[8]
- Bridging the gap between formal ontologies developed with languages like OWL and the rather informal way how knowledge is shared on the Web

Finally, a summary is provided in Sect. 13.5.

13.1 Data Model

RDF (Resource Description Framework) is a domain-independent data model that provides the main building blocks for knowledge graphs. RDFS (RDF Schema) provides a simple mechanism for capturing the semantic model of a domain and complements the primitives provided by the RDF data model. In the following, we will present:

- The basic graph-based data model of the Semantic Web, i.e., RDF
- Its enrichment by RDFS defining a simple schema language for this data model
- Several ways to provide them with a Web-oriented syntax
- Some tool support
- A summary

13.1.1 The Resource Description Framework (RDF)

Before diving into the details of RDF, let us first talk about its motivation and why existing languages for the Web, such as HTML and XML, were insufficient. *HTML* does not have enough semantics as its main purpose is to structure a Web document with a special markup for presentation purposes. Although one of the purposes of *XML* is to enable data exchange, XML documents are difficult to integrate if they are from different sources as their tags are not about expressing semantics but document structure. Let us consider the following information: "Umutcan Serles is teaching the Semantic Web course." XML can represent this information in many ways; we list two of them in Fig. 13.1. The major issue here is that the tags have no explicit meaning and depend on linguistic structures and how the tags are nested.[9]

[7] https://www.w3.org/2004/02/skos/. The prefix *skos* is used for the SKOS namespace.
[8] See Sect. 18.1.4.
[9] See Sect. 5.3 for more details.

```
<course name="Semantic Web" lecturer="Umutcan Serles" />

<course name="Semantic Web">
        <lecturer> Umutcan Serles </lecturer >
</course>
```

Fig. 13.1 Different representations in XML [Adapted from Antoniou et al. (2012)]

With its hypertext model, the Web forms a graph linking documents or subdocuments. Unsurprisingly, the data model that was preferred for metadata on the Web is graph-based. The reflection of how the documents and their linking are semantically described is also best done with a language that has a graph-based modelling approach.

RDF is a graph-based data model developed in 1997[10] as an abstract *metadata exchange language*. In 2004,[11] it was formally defined with semantics and became the data model for the Semantic Web as a W3C recommendation. The latest RDF specification is from 2014[12] and was published as RDF 1.1. In this section, we cover the latest RDF 1.1 specification.

The core of RDF is the notion of *resource*. A resource can be anything, for example:

- A hotel room
- A person
- A document (e.g., Web page)
- A piece of text (a literal value)
- A date (a literal value)

A resource can have a unique identifier in the form of an IRI,[13] or it can be a literal value. A book can be identified with a URN like urn:isbn:978-0451524935 and a person with an IRI like http://www.fensel.com. IRI (Internationalized Resource Identifier) is a generalized version of URI that allows a larger variety of Unicode characters. Every URI is also an IRI; however, the vice versa is not necessarily true. Nevertheless, there is a partial mapping between IRIs and URIs,[14] and the IRIs we use in this section also comply with the URI syntax.

In terms of the data model, a statement in an RDF graph or dataset is a *triple* in the form of:

[10] Initial RDF spec: https://www.w3.org/TR/WD-rdf-syntax-971002/

[11] RDF 1.0: https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[12] RDF 1.1: https://www.w3.org/TR/rdf11-concepts/. Section 13.1 contains content from this specification and RDF Primer https://www.w3.org/TR/rdf11-primer/. Copyright © 2003–2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[13] https://www.rfc-editor.org/rfc/rfc3987

[14] https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#dfn-iri

```
<subject> <predicate> <object>
```

The *subject* and the *object* represent two related resources. The *predicate* represents the nature of their relationship. The relationship is directional, namely, from the subject to the object. RDF offers *only* binary predicates. An example triple would be:

```
ex:jake ex:fatherOf ex:bill.
```
[15]

Three types of RDF terms can be used for building triples. These are IRIs, literals, and blank nodes.

As mentioned earlier, *IRIs* are RDF terms used to uniquely identify resources. More precisely, an IRI *denotes* a resource. Using dereferenceable IRIs is not mandatory but encouraged to provide a representation of the denoted resource when needed.[16] This way, the intended meaning of the IRI can be conveyed better.

Literals can only appear in the object position of a triple. A literal consists of three elements:

- A lexical form, which is a Unicode string
- A datatype IRI, which is an IRI identifying a datatype that decides how the lexical form is mapped to a literal value
- A non-empty language tag, if and only if a literal has the type *xsd:langString*

Below, two triples with literals in the object position are shown.

```
ex:harry   ex:age       "14"^^xsd:integer .
ex:harry   ex:hasName   "Harry Potter"@en .
```

The first one specifies that the value "14" has the datatype *xsd:integer*. RDF reuses the XML Schema datatypes[17] for literals. A subset of these datatypes are shown in Table 13.1. The second triple has a language tag "*en*" which indicates it is an *xsd:langString* literal in English.

Additionally, the RDF vocabulary offers two more datatypes:

- *rdf:HTML* for literals with HTML markup
- *rdf:XMLLiteral* for literals with XML content

A datatype consists of:

[15] *ex* is the prefix for the namespace http://example.org/ and used for examples throughout this chapter.

[16] Remember the Linked Data principles in Chap. 9.

[17] The *xsd* prefix is used for XML Schema datatypes namespace.

Table 13.1 A sample of XSD datatypes supported by RDF (a full list can be found on https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#xsd-datatypes)

Datatype	Value space
xsd:string	Character strings (but not all Unicode character strings)
xsd:Boolean	True, false
xsd:decimal	Arbitrary-precision decimal numbers
xsd:integer	Arbitrary-size integer numbers
xsd:double	64-bit floating point numbers incl. ±Inf, ±0, NaN
xsd:float	32-bit floating point numbers incl. ±Inf, ±0, NaN
xsd:date	Dates (yyyy-mm-dd) with or without timezone
xsd:time	Times (hh:mm:ss.sss…) with or without timezone
xsd:dateTime	Date and time with or without timezone

Lexical space: {"true", "false", "1", "0"}

Value space: {true, false}

Lexical-to-value mapping: {<"true", true>, <"false", false>, <"1", true>, <"0", false> }

Fig 13.2 *xsd:boolean* datatype definition

- *A lexical space*, a set of Unicode strings
- *A value space*
- A lexical-to-value mapping, a set of pairs whose first element is from the lexical space and second element is from the value space of the datatype

The XML Schema datatype *xsd:boolean* is defined as shown in Fig. 13.2:[18]

The third element, *blank nodes*, can appear in the subject and object position of a triple. Blank nodes are disjoint from IRIs and literals. They are typically used for unnamed objects or describing complex constructs. In general, a blank node corresponds to an *existential* variable in logic. Representation of blank nodes is syntax- or implementation-dependent. Below is an example with three triples. The first triple has a blank node as object which is used to represent two statements with first and last name that comprise a name value.

```
ex:harry          ex:hasName      _:harrysname .
_:harrysname      ex:firstName    "Harry"^^xsd:string .
_:harrysname      ex:lastName     "Potter"^^xsd:string .
```

[18] https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-Datatypes

Table 13.2 An overview of RDF terms and their allowed positions in a triple

	IRI	Literal	Blank node
Subject	X		X
Predicate	X		
Object	X	X	X

The three kinds of RDF terms are allowed in different positions in a triple; see also Table 13.2:

- The *subject* of a triple can be an IRI or a blank node.
- The *predicate* of a triple must be an IRI.
- The *object* of a triple can be any of the three types of terms.

In some cases, resources in an RDF dataset need to be grouped in some way. RDF provides the containers for this purpose. There are three types of containers (Antoniou et al. 2012):

- *rdf:Bag*: unordered container of items (e.g., "The event is attended by Donald, Alexandra, and Ted")
- *rdf:Seq*: ordered container of items (e.g., "The document is first edited by Max and then by Jason")
- *rdf:Alt*: a container of alternative items (e.g., "The source code for the application may be found at git1.example.org, git2.example.org, or git3.example.org")

Although they have some use cases, containers also come with some limitations. First, the semantics of containers is only given in the form of "intended semantics," and their formalization is quite limited. Second, due to the open-world semantics of RDF, it is not clear how to specify that a container has a closed or an even finite number of elements (see Sect. 14.2 for further discussion).

A set of RDF triples comprise an *RDF graph*. Therefore, they are often presented as graphs.[19] An RDF graph is a labelled, directed graph where nodes represent resources (IRI or literal) and blank nodes and edges represent predicates that connect two resources denoted by the subjects and objects of triples. Figure 13.3 shows an example RDF graph.

The support for binary predicates in RDF and its natural consequence, that is, the triple-based model, has an important side effect. It is not straightforward to make statements about statements in RDF, although it can be crucial for many reasons (e.g., provenance, signing a set of RDF triples for security). Fortunately, there were several ways developed to work around this limitation. One example is the standard RDF reification, where RDF statements are turned into resources, which allows identifying a statement with an IRI.[20] This IRI can then be used as a subject in a triple which effectively allows making statements about that statement.

[19] Remember that the AI community called such graphs semantic nets.

[20] A blank node can also be used, if the identity of the statement is not important.

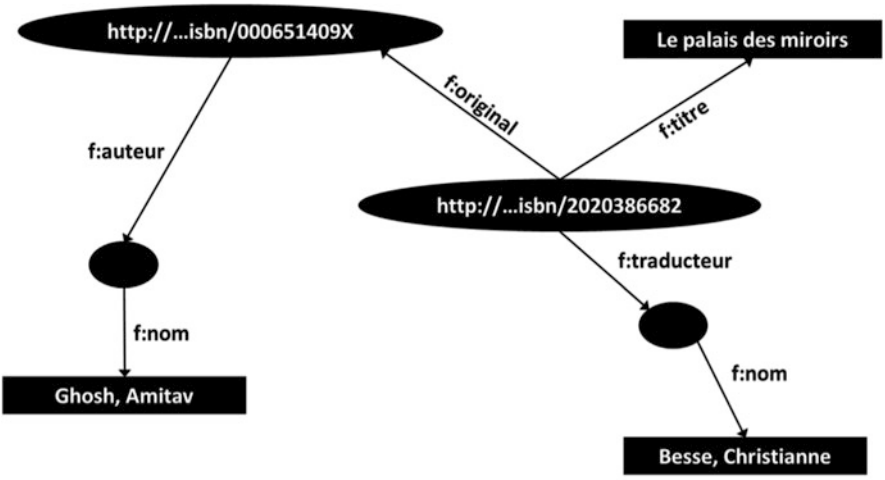


Fig. 13.3 An example RDF graph that describes a book instance (Figure taken from W3C (Herman 2012)—last accessed on 04.04.2023. The content is distributed under CC-BY 3.0)

```
ex:fredsStatement   rdf:type      rdf:Statement .
ex:fredsStatement   rdf:subject   ex:harry .
ex:fredsStatement   rdf:predicate ex:hasName .
ex:fredsStatement   rdf:object    "Harry Potter .

ex:fred             ex:claims     ex:fredsStatement
```

Fig. 13.4 Reification of "Fred claims that Harry's name is Harry Potter"

Consider the following statement: "Fred claims that Harry's name is Harry Potter." Here, Fred makes a statement about another statement. With the standard RDF reification, such a statement may look like Fig. 13.4.

Another way of implementing reification is the named graphs (Carroll et al. 2005) approach. Named graphs are a means for grouping triples in an RDF graph into subgraphs. Named graphs can be effectively implemented via quads, an extension of the triple-based model of RDF to quadruples:

```
<subject> <predicate> <object> <graph>
```

<*graph*> is an IRI that identifies the subgraph where the statement is. Using the same graph IRI for multiple statements puts them in the same subgraph. The triples in different named graphs can share IRI references but no blank nodes. Blank nodes are locally scoped and only valid for a given context. The set of blank nodes of two

Subject: https://mindlab.ai/tvb-mayrhofen/intermaps/10004768
Predicate and **Object**: <http://schema.org/name> "Fun Ride Penken" .
Graph: https://mindlab.ai/tvb-mayrhofen/intermaps .

Subject: <https://mindlab.ai/tvb-mayrhofen/intermaps/10004768>
Predicate: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
Object: http://schema.org/SportsActivityLocation
Graph: <https://mindlab.ai/tvb-mayrhofen/intermaps>.

Fig. 13.5 A simple named graph with two statements

(Sub-)Graph: <https://mindlab.ai/tvb-mayrhofen/intermaps>
Predicate and **Object**:
 # Defines the creator of the named graph
<http://schema.org/creator> <https://mindlab.ai/organization/mindlab>;
Defines the location covered by the named graph
<http://schema.org/spatialCoverage> <https://mayrhofen.at>

Fig. 13.6 Two statements over a named graph

graphs is disjoint. Figure 13.5 shows two statements about an object contained by a named graph.

A graph IRI can also be used as the subject in a triple. This allows making statements about subgraphs. The set of statements in Fig. 13.6 demonstrates how named graphs can achieve reification. For example, the two statements in Fig. 13.5 have https://mindlab.ai/tvb-mayrhofen/intermaps as their graph IRI. In Fig. 13.6, this IRI is used in the subject position in a set of triples that effectively describe the group of statements that are in the named graph identified with the IRI. Note that the IRI used in the $<graph>$ position identifies a subgraph of an RDF graph where a set of triples are contained. Therefore, any property value assertion made on $<graph>$ is defined on the subgraph and not necessarily on individual statements.[21]

A more recent approach for reification is *RDF-Star* (Hartig and Champin 2021). RDF-Star provides a compact syntax for specifying nested triples. It is still being actively developed by a W3C Working Group.[22] Several triplestore implementations are already supporting it.[23]

RDF provides a simple but useful data model for representing data on the Web, with very few built-in modelling primitives. Aside from types and certain properties used for reification, containers, and datatypes, RDF provides:

- *rdf:type* for specifying the instance relationship between a resource and a class
- *rdf:Property* for specifying that an IRI denotes a property

[21] Nevertheless, for practical purposes, a named graph with a single statement may be viewed as a reified statement (Carroll et al., 2005).

[22] https://www.w3.org/groups/wg/rdf-star

[23] See Chap. 19 for more details.

Table 13.3 RDFS modelling primitives and their implementation

Modelling primitive	RDFS term	Example
Class	rdfs:Class	<ex:Student, rdf:type, ex:rdfs:Class>
Class hierarchy	rdfs:subClassOf	<ex:Student, rdfs:subClassOf, ex:Person>
Class relationship	rdfs:domain rdfs:range	<ex:hasName, rdfs:domain, ex:Person> <ex:hasName, rdfs:range, xsd:string>
Property hierarchy	rdfs:subPropertyOf	<ex:hasFather, rdfs:subPropertyOf, ex:hasParent>

Instantiation (*rdf:type*) of classes (and even classes itself) and definition of properties (*rdf:Property*) are modelling primitives that are schema elements. This results in a slightly improper layering of a schema language over a data modelling language.

In the next section, we introduce the language for creating schemas for RDF, namely, *RDF Schema (RDFS)*, that allows the definition of classes, properties, and relationships involving them.

13.1.2 RDF Schema (RDFS)

RDF allows the instantiation of types. The following triple is the creation of an ex: *Student* instance called *ex:harry* as follows:

```
<ex:harry, rdf:type, ex:Student>
```

What is an "*ex:Student*"? RDF allows us to express statements but does not provide many modelling primitives to give a formal meaning to them. We know that "*ex:Student*" identifies a concept or a class, but this is only implicitly defined in RDF.

We need a language for defining classes (*ex:Student* is a class), class hierarchies ("*ex:Student* is a subclass of *ex:Person*"), properties that define the relationship between classes (*ex:Person* is in the domain of a property *ex:hasName*, and *xsd:string* is the range of that property), and hierarchies of properties (*ex:hasFather* is a subproperty of *ex:hasParent*). RDFS[24] is such a language. Table 13.3 shows how RDFS implements its modelling primitives.

A class in RDFS is associated with a set of instances that are conceptually cohesive in certain aspects (also called the extension of a class).[25] For example,

[24] This section contains content from RDF Schema 1.1 Specification https://www.w3.org/TR/rdf-schema/. Copyright © 2004–2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[25] Note that RDFS makes a difference between classes and the resources in their extensions (set of instances). Two distinctly named RDFS classes may have exactly the same set of instances, but those instances may be described with different properties. See also https://www.w3.org/TR/rdf-schema/#ch_classes.

the class of students groups a set of instances, e.g., people who enrolled at an educational institution. A class is defined as an instance of *rdfs:Class*.

The classes in RDFS can be organized hierarchically. This implies a subset relation between the extensions of two classes. For example, the class of students can be defined as a subclass of the class of persons. This implies that all instances of the class *Student* are also the instances of the class *Person*.

A property in RDF defines a relationship between two resources, which creates a set of *<subject, object>* pairs that is the extension of that property. The instances of two classes may be connected via a property. The mechanism for this is defined by RDFS via domain and range of properties. The domain (*rdfs:domain*) of a property implies that every resource the property is used on is an instance of the type defined in the domain. Similarly, the range (*rdfs:range*) of a property implies that every resource that is a value of the property is an instance of the type defined in the range.[26] For example, a property called *ex:hasName* can have the class *ex:Person* in its domain and *xsd:string* in its range. This implies that any resource described with the ex:*hasName* property is an instance of the class ex:*Person*. Also, every value of the ex:*hasName* property is an instance of *xsd:string*.

Like classes, properties can also be organized hierarchically. This creates a subset relationship between the extensions of two properties. For example, if ex:*hasFather* is a subproperty of *ex:hasParent*, then every subject and object that is connected with the *ex:hasFather* property is also connected with the *ex:hasParent* property. The formal definitions of RDFS semantics are presented in Chap. 14.

Additional to the core properties mentioned above, RDFS also provide various annotation properties that can be used to attach additional information to a resource:

- *rdfs:comment* is used to specify a human-readable description of a resource. This property is typically used to provide additional clarification about the intended meaning of a resource, e.g., *ex:Person*, *rdfs:comment*, *"A person is a human being."* Multilingual definitions are possible via the language tagging mechanism of RDF.
- *rdfs:label* is used to specify a human-readable version of the resource name, e.g., *ex:Person, rdfs:label, "Person."* Multilingual definitions are possible via the language tagging mechanism of RDF.
- *rdfs:seeAlso* is used to indicate another resource that might provide additional information about the resource being described. The nature of the linked resource is not clearly specified by RDFS. For example, it can be a Web page or another RDF file. For instance, *ex:Person rdfs:seeAlso* https://dbpedia.org/page/Human specifies that more information about *ex:Person* can be found in https://dbpedia.org/page/Human.
- *rdfs:isDefinedBy* is typically used for specifying the resource that defines another resource. More concretely, a resource can be connected to the RDFS vocabulary

[26] As evident from the domain and range definitions, RDFS does not have locally defined properties on classes as in frame-based languages introduced in Chap. 2. OWL (Sect. 13.3) has some mechanisms to define properties locally on types.

that describes it with this property. *rdfs:isDefinedBy* is a subproperty of rdfs: seeAlso. For example, *dbpedia:Person rdfs:isDefinedBy* https://dbpedia.org/ specifies that the *dbpedia:Person* concept is described within DBpedia.

13.1.3 RDF(S) Serialization

RDF and RDF(S) have abstract syntax and support many concrete syntaxes as serialization formats.[27] Different ways of writing down the same RDF graph lead to the same triples and therefore are logically equivalent. Some serialization formats are as follows:

- The Turtle family of RDF languages: N-Triples, N-Quads, Turtle, and TriG
- JSON-based RDF syntax: JSON-LD
- RDFa for HTML and XML embedding, i.e., usage as Web site annotation
- RDF/XML syntax for RDF

In the following, we will introduce the serialization formats mentioned above and represent the RDF graph in Fig. 13.7 with each one of them.[28] The graph shows a *Person* instance named *Bob*, who knows *Alice* and has an interest in *Mona Lisa*.

N-Triples.[29] N-Triples is a line-based, plain text format for encoding an RDF graph. Its original intent was for creating test cases, but it became popular beyond. URIs are enclosed in angle brackets (<>). The period (.) at the end of the line signals the end of the triple. A datatype is appended to the literal through a ^^ delimiter. The datatype may be omitted when specifying a string literal. Figure 13.8 shows an example of the N-Triple serialization of the graph in Fig. 13.7. The N-Triples format can be extended with another element for *graph* to support named graphs. This format is then called *N-Quads*.[30]

Turtle.[31] The Terse RDF Triple Language (Turtle) is an extension of N-Triples. Turtle offers a trade-off between ease of writing, ease of parsing, and readability. It has the following main features:

---

[27] One could argue that syntax and the following tools section actually are issues for the implementation level. Therefore, we provide here only short summaries.

[28] Section 13.1.3 contains content including the figures taken from https://www.w3.org/TR/rdf11-primer/. © 2003–2014 World Wide Web Consortium (MIT, ERCIM, Keio, Beihang). http://www.w3.org/Consortium/Legal/2015/doc-license

[29] https://www.w3.org/TR/2014/REC-n-triples-20140225/. Copyright © 2008–2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[30] https://www.w3.org/TR/2014/REC-n-quads-20140225/. Copyright © 2008–2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[31] https://www.w3.org/TR/2014/REC-turtle-20140225/. Copyright © 2008–2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

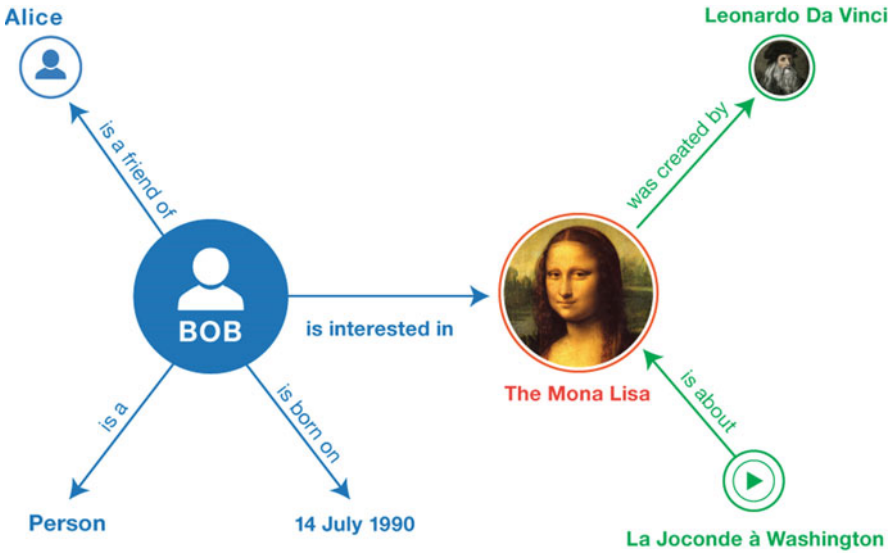


Fig. 13.7 An RDF graph about Bob, Alice, and Mona Lisa

- *BASE and Relative URIs*: Turtle syntax allows the definition of a base IRI that allows usage of relative IRIs. For example, in Fig. 13.9, <*bob#me*> is equivalent to <http://example.org/bob#me> after expanding with the defined base URI.
- *PREFIX and prefixed names*: Similar to the base IRI, Turtle syntax supports the definition of namespaces and their prefixes. For example, in Fig. 13.9, given the prefix wd:, the short IRI *wd:Q12418* can be expanded to https://wikidata.org/entity/Q12418.
- *Predicate lists separated by ";"*: As a syntactic convenience for compactness, different predicates attached to the same subject can be split with a semi-colon (;). For example, in Fig. 13.9, all predicates attached to <*bob#me*> are split via a semi-colon.
- *Object lists separated by ","*: Similar to the shortcut above, the values of the same predicate on a subject can be split via a comma (,). For example, if Bob knew another person next to Alice, the URI of this person and <*alice#me*> could be written separately with a comma, without repeating the *foaf:knows* predicate again.
- The token "a" as a shorthand for *rdf:type*: The keyword "a" can be used as a shortcut for instantiation. For example, in Fig. 13.9, <*bob#me*> is asserted as an instance of *foaf:Person.*

TriG.[32] TriG is a plain text format for serializing named graphs. The TriG syntax provides a compact alternative for the serialization of named graphs. TriG is an

[32] https://www.w3.org/TR/2014/REC-trig-20140225/. Copyright © 2008–2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

```
<http://example.org/bob#me> <http://www.w3.org/1999/02/22-
rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .

<http://example.org/bob#me> <http://xmlns.com/foaf/0.1/knows>
<http://example.org/alice#me> .

<http://example.org/bob#me> <http://schema.org/birthDate>
"1990-07-"^^<http://www.w3.org/2001/XMLSchema#date> .

<http://example.org/bob#me>
<http://xmlns.com/foaf/0.1/topic_interest>
<http://www.wikidata.org/entity/Q12418> .

<http://www.wikidata.org/entity/Q12418>
<http://purl.org/dc/terms/title> "Mona Lisa" .

<http://www.wikidata.org/entity/Q12418>
<http://purl.org/dc/terms/creator>
<http://dbpedia.org/resource/Leonardo_da_Vinci> .

<http://data.europeana.eu/item/04802/243FA8618938F4117025F1
7A8B813C5F9AA4D619> <http://purl.org/dc/terms/subject>
<http://www.wikidata.org/entity/Q12418> .
```

Fig. 13.8 N-Triple representation of the graph in Fig. 13.7

extension of Turtle. For example, additional to all Turtle features, the triples can be put in a block that can be identified with an IRI or blank node, which represents the named graph that consists of the triples in that block. The GRAPH keyword is optional; however, it can be used to improve the compatibility with SPARQL (Sect. 13.2.1). The named graphs in Fig. 13.10 can be shown in TriG syntax as shown in Fig. 13.11. In this example, http://example.org/bob and https://www.wikidata.org/wiki/Special:EntityData/Q12418 represent two named graphs. The triples outside the GRAPH blocks describe the named graph identified with http://example.org/bob.

JSON-LD.[33] JSON-LD provides a JSON[34] syntax for RDF graphs and datasets. JSON is a hierarchical data exchange format. JSON supports two main data structures: a collection of key-value pairs and an ordered list of values. The former in JSON is called an object. The key-value pairs are placed in curly brackets {} and separated by a comma. The latter is called an array. An array consists of values that are separated with a comma between brackets []. A value can be a string, a number, a Boolean value, an object, or an array. As you see, the JSON syntax is quite generic, and there is no distinct data structure for IRIs or any other datatype other than string, Boolean, or numbers. Therefore, JSON-LD extends the JSON syntax with special

[33] https://www.w3.org/TR/json-ld/. Copyright © 2010–2020 W3C® (MIT, ERCIM, Keio, Beihang). W3C liability, trademark, and permissive document license rules apply.

[34] https://www.json.org/json-en.html

```
BASE  <http://example.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX schema: <http://schema.org/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX wd: <http://www.wikidata.org/entity/>

<bob#me>
      a foaf:Person ;
      foaf:knows <alice#me> ;
      schema:birthDate "1990-07-04"^^xsd:date ;
      foaf:topic_interest wd:Q12418 .

   wd:Q12418
      dcterms:title "Mona Lisa" ;
      dcterms:creator <http://dbpedia.org/resource/Leonardo_da_Vinci> .

   <http://data.euro-
peana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D6
19>
```

Fig. 13.9 Turtle representation of the graph in Fig. 13.7

properties to enable RDF support. The following are the major syntactic tokens and keywords supported:

- @*base*: used to set the base IRI. The base IRI works as same as in Turtle syntax.
- @*context*: used to provide additional heuristics that is needed to make JSON syntax eligible for representing RDF. The value of the @*context* property is a context file that mainly determines how the keys and values will be expanded when a JSON-LD document is parsed into an RDF graph. Each key represents a property, and whether their values are literal or should be expanded to IRIs is determined by the context file.[35]
- @*id*: a special property that is used to identify a node in an RDF graph. Any object without an IRI in the @*id* value is treated as blank nodes. An explicit blank node identifier can be also defined (starting with an underscore (_)).
- @*type*: used to set the type of a node or the datatype of a typed value. Its value corresponds to the value of *rdf:type* property in a triple.

There are many other features, such as named graph and RDF container support, which can be found in the JSON-LD specification. JSON-LD developers also

[35] See https://json-ld.org/spec/latest/json-ld/#the-context for more detailed description of what is supported in a context file.

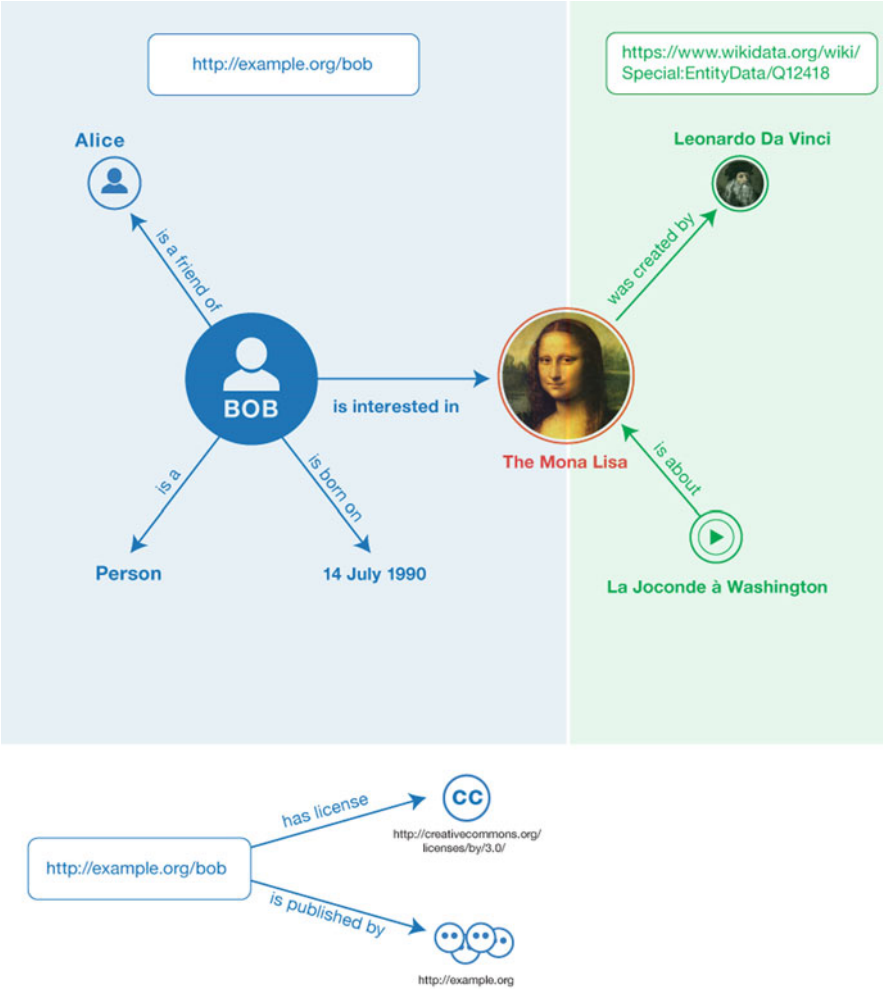


Fig. 13.10 The example graph in Fig. 13.7 is organized in two named graphs. The lower part of the image shows the triples attached to the named graph containing triples about Bob

provide a playground for trying different features of the language.[36] Figure 13.12 shows the JSON-LD representation of the graph in Fig. 13.7.

RDFa.[37] RDFa is used to embed RDF data within HTML and XML documents. It offers the following attributes and more:

[36] https://json-ld.org/playground/

[37] https://www.w3.org/TR/rdfa-primer/. Copyright © 2010–2015 W3C® (MIT, ERCIM, Keio, Beihang). W3C liability, trademark, and document use rules apply.

```
BASE   <http://example.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX schema: <http://schema.org/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX wd: <http://www.wikidata.org/entity/>

GRAPH <http://example.org/bob>
 {
  <bob#me>
     a foaf:Person ;
     foaf:knows <alice#me> ;
     schema:birthDate "1990-07-04"^^xsd:date ;
     foaf:topic_interest wd:Q12418 .
 }

GRAPH <https://www.wikidata.org/wiki/Special:EntityData/Q12418>
 {
  wd:Q12418
     dcterms:title "Mona Lisa" ;
     dcterms:creator <http://dbpedia.org/resource/Leonardo_da_Vinci> .

    <http://data.euro-
peana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619>
      dcterms:subject wd:Q12418 .
 }

<http://example.org/bob>
   dcterms:publisher <http://example.org> ;
   dcterms:rights <http://creativecommons.org/licenses/by/3.0/> .
```

Fig. 13.11 TriG representation of the graph in Fig. 13.10

- *resource*: specifies the URI about which RDF statements can be made within this HTML element
- *property*: specifies an RDF property, e.g., *foaf:knows*
- *typeof*: specifies the instantiation relationship (via *rdf:type*)
- *prefix*: specifies namespace prefixes in a similar way as the Turtle prefixes

RDFa attributes are directly attached to HTML tags where the content resides. This has the advantage that updating the Web site content also updates the RDF data most of the time.

RDF/XML. RDF/XML uses XML syntax for serializing RDF data. It is the original serialization format that was used since the inception of RDF. It allows the usage of existing XML-based tools and parsers reducing the entry barrier for tool developers and users. Note that even though the syntax is XML, the data is still represented independently from the structure of the document. RDF/XML lost

```
{
  "@context": "example-context.json",
  "@id": "http://example.org/bob#me",
  "@type": "Person",
  "birthdate": "1990-07-04",
  "knows": "http://example.org/alice#me",
  "interest": {
    "@id": "http://www.wikidata.org/entity/Q12418",
    "title": "Mona Lisa",
    "subject_of": "http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619",
    "creator": "http://dbpedia.org/resource/Leonardo_da_Vinci"
  }
}
```

Fig. 13.12 JSON-LD representation of the RDF graph in Fig. 13.7

popularity in recent years due to its syntactic overhead, which hampers human readability.

13.1.4 Tools

There is a plethora of RDF tools. Covering them all in detail would require another book. We can roughly put these tools in categories like *editors*, *browsers*, *triplestores*, and *validators*. We refer the readers to online compilations for a large set of tools in a wide range of categories.[38]

Editing RDF does not technically require a tool more advanced than a text editor. However, in recent years, many integrated development environments (IDEs) like Eclipse[39] and Visual Studio Code[40] had extensions that support RDF syntax specifically with syntax highlighting and autocompletion.

Browsing RDF is typically done via Web-based tools that extract RDF data from Web pages or directly parse a given RDF file. A prominent example of such a tool is the OpenLink Structured Data Sniffer,[41] which runs an extension in major browsers and presents extracted RDF data from Web pages. Moreover, many triplestores and graph databases provide interfaces for browsing RDF data.[42]

Triplestores are perhaps the most well-established software products in the RDF ecosystem. They provide hosting, browsing, and querying mechanisms for RDF data (more on this later in Chap. 19).

[38] For example, https://github.com/semantalytics/awesome-semantic-web

[39] For example, https://github.com/AKSW/Xturtle

[40] For example, https://marketplace.visualstudio.com/items?itemName=stardog-union.vscode-stardog-languages

[41] https://osds.openlinksw.com/

[42] An example is the web client of Atom Graph: https://github.com/AtomGraph/Web-Client.

Finally, if you are unsure about the syntactic *validity* of your RDF document, you can always use one of the RDF validators.[43]

13.1.5 Summary

RDF is a graph-based data model developed as a standard data model for the Semantic Web. Like many other technologies in the Semantic Web Stack, RDF was adopted by knowledge graphs as one of the most widespread data models.

RDF Schema extends RDF with a few modelling primitives to enable knowledge graph developers to define a schema to describe the RDF data.

In this section, we introduced these two languages from an epistemological point of view. RDF provides mechanisms to represent resources on the Web and connect them with binary relationships, which form triples. A resource can be anything, for example, a hotel room, a person, or a Web page. RDFS provides modelling primitives to describe RDF data, such as classes, domain and range definitions for properties, and class and property hierarchies. Interestingly, property definitions and instantiation relationships are defined by RDF and not RDFS. This situation makes the two languages strongly coupled and makes it challenging to discuss these two languages properly in a layered manner.

This is also why working with RDF and RDFS typically goes together, and the syntaxes and tools apply to both languages. Although arguably belonging to the implementation level, we also briefly introduced various syntaxes and tools.

Despite its simplistic model and very limited semantics, many knowledge graphs adopt RDF(S) as a modelling formalism. Still, there are cases where more complex semantics is needed. We will discuss this in Sect. 13.3.

13.2 Data Retrieval and Manipulation

A knowledge graph needs to be queried for retrieval and manipulation to be used by humans and applications. In this section, we will introduce two languages for this purpose from an epistemological point of view. The first is the *SPARQL query language* for RDF data retrieval and manipulation. The second one is the *Shapes Constraint Language (SHACL)*, which provides a declarative way to create constraints over RDF data. Finally, we conclude the section with a summary.

[43] For example, W3C Validator (http://www.w3.org/RDF/Validator/)

13.2.1 SPARQL

SPARQL is the query language for RDF datasets and a W3C recommendation. It supports federated queries, which allow straightforward access to distributed RDF datasets over HTTP. With SPARQL, RDF data can be queried and manipulated. In the remainder of this section, we first start with the queries for data retrieval and then cover the queries for data manipulation. Then we discuss the concept of subqueries. For both retrieval[44] and manipulation[45] queries, we use the SPARQL 1.1 Specification. We cover the core elements of the SPARQL specification; more details can be found in the specification documentation. Finally, we provide a brief collection of tools and applications.

13.2.1.1 Data Retrieval

Figure 13.13 shows the anatomy of a SPARQL query for data retrieval. In detail, we will explain with examples from top to bottom prefixes, query result clause, dataset definition, query pattern, and query modifiers.[46]

Prefix declarations.[47] Prefixes define a shorthand for namespaces. A prefix in the query connects the IRI of a namespace to the local part of an IRI to build an identifier for a resource. Prefix declaration can also contain an empty prefix (denoted with just ":").

The example in Fig. 13.14 defines the prefix *ex* for the URI http://example.org/StatesOfAustria.ttl#. During the query execution, for example, *ex:State* would be expanded to http://example.org/StatesOfAustria.ttl#State.

Query result clauses: The query result clause specifies the results expected from a SPARQL query. There are four types of result clauses:

- SELECT
- ASK
- CONSTRUCT
- DESCRIBE

[44] Section 13.2.1.1 contains content from https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. Copyright © 2013 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[45] Section 13.2.1.2 contains content from https://www.w3.org/TR/sparql11-update/. Copyright © 2013 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[46] The examples involve queries over a fictive dataset containing data about Austrian states and their districts. The examples are inspired by the examples from Käfer (2021) about boroughs of Berlin.

[47] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#prefNames

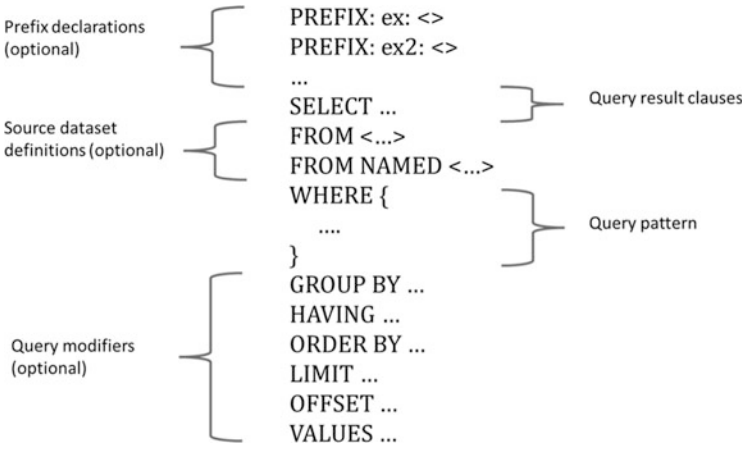


Fig. 13.13 The structure of a SPARQL query (Adapted from https://www.slideshare.net/LeeFeigenbaum/sparql-cheat-sheet)

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT ?district
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    ?s a ex:State.
    ?district ex:district ?s .

} ORDER BY ?district
```

Fig. 13.14 An example query with PREFIX declaration highlighted

SELECT clause.[48] The SELECT clause returns all or a projection of the solution mappings, based on the query pattern in the WHERE clause of the query.

A *solution mapping* is a partial mapping between the variables (a string prefixed with a question mark) in the query and RDF terms. A solution mapping consists of variable-value pairs which are also called variable bindings.

In Fig. 13.15, the query returns only the bindings involving the *?district* variable to the user as it is the only variable in the SELECT clause.

The solutions returned by the SELECT clause can be modified with the DISTINCT[49] keyword. This keyword eliminates the duplicate solution mappings in the results. For instance, different states may have districts with the same name. The DISTINCT keyword ensures that only unique values are presented to the user.

[48] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#select

[49] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#modDuplicates

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT ?district
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    (a query pattern)
}
```

Fig. 13.15 An example query with the SELECT clause highlighted

```
SELECT ?state (COUNT (?district) AS ?districtCount)
WHERE {
        ?district :district ?state.

} GROUP BY ?state
```

Fig. 13.16 An example query with the COUNT clause highlighted

The results returned by the SELECT clause can also be aggregated. An aggregation function computes a value based on the groups of variable bindings. The grouping can be done with the GROUP BY keyword.[50] The following aggregation functions are commonly used:[51]

- COUNT: gives the number of bindings
- SUM: gives the sum of the values bound
- MIN: gives the minimum of the values bound
- MAX: gives the maximum of the values bound
- AVG: gives the arithmetic mean of the values bound to a given variable

SUM and AVG aggregations only work with numerical values properly.

The example in Fig. 13.16 first groups all districts by their states and then counts all *?district* bindings per *?state* binding.

ASK clause.[52] The ASK clause returns true if there is a solution mapping given the patterns in the WHERE clause; otherwise, it returns false (Fig. 13.17).

CONSTRUCT clause.[53] The CONSTRUCT clause returns an RDF graph given a template in the form of a graph pattern. The template ideally contains a subset of the variables in the WHERE clause. The graph is built by replacing the variables in the template with the bindings of the same variables in the WHERE clause. After the substitution of variables, any illegal triples or triples with unbound variables in the

[50] GROUP BY keyword is explained in more detail below where we talk about query modifiers.

[51] A full list can be found in SPARQL specification: https://www.w3.org/TR/sparql11-query/#aggregates.

[52] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#ask

[53] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#construct

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
ASK
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    (a query pattern)

}
```

Fig. 13.17 An example query with the ASK clause highlighted

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
PREFIX : <http://example12.org/StatesOfAustria.ttl#>

CONSTRUCT { ?state :hasDistrict ?x }
WHERE {
?x ex:district ?state
}
```

Fig. 13.18 An example query with the CONSTRUCT clause highlighted

template are not included in the resulting graph. A typical use case for CONSTRUCT queries is to infer new statements from an RDF dataset.[54] It can be used to translate an RDF graph into another one with different terminology. The example in Fig. 13.18 creates an RDF graph that contains all districts connected to their states with the *ex:hasDistrict* property.[55]

DESCRIBE clause.[56] The DESCRIBE clause returns an RDF graph that is related to a given IRI. The main purpose of a DESCRIBE query is to give a "description" of a resource, given its URI. The meaning of DESCRIBE queries is not a part of the SPARQL specification; therefore, what the description of an instance contains is decided on the implementation level. There is no guarantee that different triplestores would return the same results for the instance. The example in Fig. 13.19 *may* return all triples that have *:Tyrol* in its subject position.

Source dataset definition.[57] A source dataset definition is an optional part of the query. It is only needed if the RDF dataset is not known by the SPARQL client or, for example, named graphs need to be queried. There are two ways to define source datasets with different purposes:

[54] Basically mimicking rule-based inference

[55] Here, we mimic a rule stating that *ex:hasDistrict* is an inverse property of *ex:district.*

[56] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#describe

[57] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#rdfDataset

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>

DESCRIBE ex:Tyrol
```

Fig. 13.19 An example query with the DESCRIBE clause highlighted

- **FROM**: The specified graphs are merged into the default graph. The default graph (of an RDF dataset) does not have a name, and it is the graph in which query patterns are matched by default if they are not scoped with a named graph.
- **FROM NAMED**: The query is evaluated on the specified named graphs separately. The value following FROM NAMED is the IRI of a named graph. During the query time, only one graph is active at any given time (Fig. 13.20).

The example in Fig. 13.20 identifies three different sets of graph patterns that run on different graphs in an RDF dataset:

- On a default graph that is a union of two named graphs, ex:TyrolGraph and ex:SalzburgGraph (a)
- On a named graph called ex:CorinthiaGraph (b)
- Separately in each named graph that are specified via FROM NAMED (c)

Query patterns. Query patterns specify what we want to query. They are placed in the WHERE block in a SPARQL query. A query pattern consists of a set of graph patterns, and graph patterns are built with triple patterns.

Graph patterns.[58] A graph pattern is a set of triple patterns**.** A query pattern is evaluated by matching its graph patterns. There are two main types of graph patterns:

- *Basic graph patterns* are a set of triple patterns that should match sets of triples in the RDF dataset**.**
- *Group graph patterns* are a set of graph patterns that must all match with a set of triples in the RDF dataset.

A *triple pattern*[59] is a triple *(s,p,o)*:

- s is a variable, a literal,[60] an IRI, or a blank node.
- p is a variable or an IRI.
- o is a variable, an IRI, a literal, or a blank node.

A variable is a string prefixed with a question mark "?". IRI and literal values are defined the same as in RDF. Absolute URIs are written between <>. Literals are written in double quotes "". A blank node is either represented with an arbitrary label

[58] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#GraphPattern

[59] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#sparqlTriplePatterns

[60] Note that although RDF does not allow literals in the subject position, SPARQL does not make such a restriction.

```
PREFIX ex: http://....
SELECT ....
FROM ex:TyrolGraph
FROM ex:SalzburgGraph
FROM NAMED ex:CorinthiaGraph
FROM NAMED ex:ViennaGraph
FROM NAMED ex:BurgenlandGraph
WHERE {
  (a)<graph patterns looked in the default graph>
  GRAPH ex:CorinthiaGraph {
  (b)<graph patterns that are looked in ex:CorinthiaGraph>
  }
  GRAPH ?g {
  (c)<graph patterns looked in each named graph>
  }
}
```

Fig. 13.20 An example query with the data source specifications highlighted

or a placeholder encoded with brackets [ ], which is turned into a unique blank node label during runtime. The blank node labels behave like variables and do not identify a specific node in the graph.

The set of triple patterns in a basic graph pattern can be viewed as connected with logical conjunction. This means, unless otherwise modified, all triple patterns must match to put the variable bindings in a solution mapping. The example in Fig. 13.21 contains a basic graph pattern in the query pattern that consists of two triple patterns. The query must match this basic graph pattern to return results.

The example in Fig. 13.22 shows two group graph patterns, each of which contains one basic graph pattern.[61] The difference between basic and group graph patterns can be very subtle in most cases in terms of the results obtained, although the query evaluation is different. In principle, there is one basic graph pattern in Fig. 13.21 with two triple patterns, and both triple patterns must match to obtain a result. In Fig. 13.22, there are two group graph patterns with one triple, and both graph patterns should match to obtain a result. In the end, the same triple patterns are used in conjunction, so the results of both queries would be the same.

Let us see another example where it makes a difference to use a basic graph pattern or a group graph pattern. Let us add a FILTER[62] to the query in Fig. 13.21 and create the query illustrated in Fig. 13.23. This query dictates that the query should only return results if there is an IRI bound to the *?district* variable that contains the word "Inn."

Now, let us look at the query in Fig. 13.24. This time the FILTER is in the first group graph pattern; therefore, the behavior of the query is different. This query would not return any results, as the FILTER is scoped by the graph pattern it is in,

[61] This example can be also seen as a group of two basic graph patterns.

[62] The keyword is explained in detail below.

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT *
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    ?s a ex:State.
    ?district ex:district ?s .

}
```

Fig. 13.21 An example query with a basic graph pattern highlighted

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT *
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    {?s a ex:State.}
    {?district ex:district ?s .}
}
```

Fig. 13.22 An example query with two group graph patterns highlighted

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT *
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    ?s a ex:State.
    ?district ex:district ?s .
    FILTER (contains(str(?district), "Inn")).

}
```

Fig. 13.23 An example query with a basic graph pattern and a FILTER statement

and in this group graph pattern, there is no *?district* variable binding. Since the query must successfully match all graph patterns, and the first one does not return a result, the query automatically returns an empty result, even though there are districts with the word "Inn" in their IRI.[63]

The graph patterns in a SPARQL query can be combined in several ways. The *OPTIONAL*[64] keyword defines a graph pattern as optional to match. A query does not fail if this pattern does not match any results; only the variables that are supposed

[63] The difference between two queries can be seen better in their SPARQL algebra representations. Try yourself with http://sparql.org/query-validator.html. See also Sect. 14.3.

[64] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#optionals

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT *
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    {?s a ex:State.
    FILTER (contains(str(?district), "Inn")).}
    {?district ex:district ?s .}

}
```

Fig. 13.24 An example query with two group graph patterns and one FILTER within a group graph pattern highlighted

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>

SELECT ?x ?y
WHERE
{
        ?x ex:district ex:Tyrol .
        OPTIONAL { ?x ex:population ?y . }
}
```

Fig. 13.25 An example query with an optional graph pattern highlighted

to be bound in the graph patterns in the OPTIONAL block are returned empty.[65] The example in Fig. 13.25 returns all districts of Tyrol and the population of the ones that have a value for the *ex:population* property in the graph.

The *UNION*[66] keyword evaluates a set of graph patterns separately and merges the solution mappings (set union). The example in Fig. 13.26 shows the UNION of three graph patterns. The population values of *ex:Innsbruck*, *ex:Landeck*, and *ex:Schwaz* instances are separately found and put in three different sets. The sets are then merged with the set union operation in the result.

The values bound to the variables in the patterns can be further modified with *FILTER* and *BIND*. The *FILTER* keyword takes an expression that returns a Boolean value based on the variables in the patterns. A solution mapping is in the result if it makes the expression true. The filtering expression can contain functions as well as arithmetic and Boolean operators. A list of SPARQL functions that can be used can be found in the specification.[67] The example in Fig. 13.27 returns the districts of Tyrol that have a population bigger than 100,000.

[65] Akin to the LeftJoin operation in the relational algebra

[66] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#alternatives

[67] https://en.wikibooks.org/wiki/SPARQL/Expressions_and_Functions

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>

SELECT ?y
WHERE
{
        { ex:Innsbruck ex:population ?y . }
        UNION
        { ex:Landeck ex:population ?y . }
        UNION
        { ex:Schwaz ex:population ?y . }
}
```

Fig. 13.26 An example query with the UNION graph patterns highlighted

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT ?district
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    ?district ex:district    ex:Tyrol ;
              ex:population  ?population .
    FILTER(?population > 100000)
}
```

Fig. 13.27 An example query with the FILTER clause highlighted

The *BIND*[68] keyword explicitly binds a value to a variable within the query. The bound value is typically a result of an expression but can be also any RDF term. Like the FILTER expressions, the values for BIND operation can be calculated with arithmetic, Boolean operations, and functions. The bound variable can then also be used in the query result clause. The example in Fig. 13.28 shows a query that returns the population difference between Innsbruck and Kufstein. The difference is calculated within the query as the absolute value of the population difference of the two districts and then bound to *?diffPop* variable with the *BIND* keyword.

Query modifiers[69] modify the set of solution mappings. Their usage is optional in a query. We will cover the most important ones,[70] namely, ORDER BY, LIMIT, OFFSET, and GROUP BY.

The ORDER BY modifier changes the order of the solution mappings in the result. ORDER BY modifier can have a sequence of variables (order comparators) as value. The ordering happens first by the first variable in the sequence, then second, then third, etc. The ordering can be specified as ascending (with ASC order modifier)

[68] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#bind

[69] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#solutionModifiers

[70] A complete list of such modifiers can be found at https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#solutionModifiers.

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT ?diffPop
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    ex:Innsbruck ex:population ?popIbk
    ex:Kufstein ex:population ?popKu .
    BIND (abs(?popIbk-?popKu) as ?diffPop)

}
```

Fig. 13.28 An example query with the BIND statement highlighted

```
PREFIX ex: <http://example.org/StatesOfAustria.ttl#>
SELECT ?district
FROM <http://example.org/StatesOfAustria.ttl>
WHERE {
    ?district ex:district ?s .

}
ORDER BY ASC (?district)
LIMIT 10
OFFSET 3
```

Fig. 13.29 An example query with the ORDER BY, LIMIT, and OFFSET modifiers highlighted

or descending (with DESC order modifier). The result of ORDER BY is an *ordered sequence* of solution mappings.

The LIMIT modifier puts an upper bound on the number of solution mappings returned. If the number of actual solutions is greater than the specified limit, then, at most, the specified number of solutions will be returned, although a deterministic result cannot be guaranteed across triplestores.

The OFFSET modifier specifies the starting point in a sequence of solution mappings. The default OFFSET value is 0 (i.e., starting from the first solution mapping).

Note that the results after using LIMIT and/or OFFSET are not necessarily predictable unless the solution mappings are ordered with the ORDER BY modifier. The query in Fig. 13.29 returns all districts in the graph, ordered alphabetically by their IRIs. A total of ten results are returned, starting from the fourth result in alphabetical order.

The *GROUP BY* modifier is used in combination with aggregation functions in the query result clause. It specifies the solution mappings according to which the aggregated variable bindings must be grouped. The example in Fig. 13.30 shows a query where the number of districts is returned grouped by each state.

```
SELECT ?s (COUNT (?district) AS ?districtCount)
WHERE {
        ?s a ex:State.
        ?district ex:district ?s .
} GROUP BY ?s
```

Fig. 13.30 An example query with the GROUP BY modifier highlighted

13.2.1.2 Data Manipulation

Starting from version 1.1, SPARQL can be used to insert or delete triples from a graph. We cover two types of manipulation operations. These operations are *inserting* triples to or *deleting* from an RDF graph based on a condition and inserting or deleting a (sub)graph to/from an RDF graph.

Conditional insert and delete.[71] *INSERT* and *DELETE* queries can be used for adding and deleting triples from an RDF graph, respectively. These operations can also be combined in the same query (to simulate an UPDATE operation). The triples are only modified for the triples that match the pattern in the WHERE clause. The WHERE clause can be built with primitives explained above for query patterns (e.g., FILTER, OPTIONAL, BIND).

The DELETE/INSERT syntax allows shortcuts for specifying the graph where the query pattern in a WHERE clause will be evaluated (WITH and USING keywords).[72]

The example in Fig. 13.31 shows, for each instance with the *:name* value "Innsbruck," remove the triple with "Innsbruck" as the object for *:name* and add the triples with "City of Innsbruck" as the object for the *:name* property.

Inserting and deleting an RDF graph. *INSERT DATA*[73] and *DELETE DATA*[74] queries are used to add or delete a set of RDF triples (without condition) to a (named) graph. The triples should not contain any variables. If blank nodes are used while inserting, they are treated as new blank nodes and have nothing to do with the existing blank nodes in the target graph. The *DELETE DATA* operation additionally does not allow any blank nodes. The example in Fig. 13.32 shows a query that inserts three triples into an RDF graph (the default graph in this case).

[71] https://www.w3.org/TR/sparql11-update/#deleteInsert

[72] See the paragraphs about WITH and USING keywords, as well as their meaning in https://www.w3.org/TR/sparql11-update/#deleteInsert.

[73] https://www.w3.org/TR/sparql11-update/#insertData

[74] https://www.w3.org/TR/sparql11-update/#deleteData

```
PREFIX : <http://example.org/StatesOfAustria.ttl#>

DELETE { ?city :name “Innsbruck”}
INSERT { ?city :name “City of Innsbruck” }
WHERE { ?city :name “Innsbruck” }
```

Fig. 13.31 An example query DELETE/INSERT highlighted

```
PREFIX : <http://example.org/StatesOfAustria.ttl#>

INSERT DATA {
:Innsbruck  a      :District ;
      :name        "Innsbruck";
      :population  "135623"^^xsd:integer .
}
```

Fig. 13.32 An example query with the INSERT DATA clause highlighted

13.2.1.3 Subqueries

Subqueries[75] allow embedding SPARQL queries within other SPARQL queries in order to achieve complex tasks that are otherwise infeasible or impossible to achieve with a single query. Since SPARQL queries are evaluated in a bottom-up fashion, first, the inner query is evaluated, and then its results are used in the outer query. Note that only the solution mappings created based on the variables defined by the SELECT clause of the inner query are projected to the outer query. Figure 13.33 shows a SPARQL query with a subquery.

The subquery in Fig. 13.33 returns states and the average population of each state’s districts. Then the outer query returns the countries and the maximum average population they are neighboring.

13.2.1.4 Tools and Applications

As the standard query language for RDF data, tools and applications for working with SPARQL are too many to cover here comprehensively. We will, however, mention at least the categories of such tools and applications.

Tools for SPARQL can be covered in three categories, editors, engines, and validators.

- For *editing SPARQL*, any text editor would be sufficient. For more flexibility and assistance in creating queries, the integrated development environments we

[75] https://www.w3.org/TR/sparql11-query/#subqueries

```
PREFIX : <http://example.org/StatesOfAustria.ttl#>
SELECT  ?country (MAX(?avgPop) as ?maxAvgPop)
WHERE {
?country :hasBorder ?state
{
   SELECT ?state (AVG(?population) as ?avgPop)
    WHERE
    {
       ?district :district ?state .
       ?district :population ?population .
    }  GROUP BY ?state
}

} GROUP BY ?country
```

Fig. 13.33 A SPARQL query with a subquery highlighted

mentioned in Sect. 13.1.4 for RDF can also be used. There are also external SPARQL clients like YASGUI (Rietveld and Hoekstra 2017) that can be integrated with other tools (e.g., triplestores) that provide syntax highlighting and autocompletion as well as can send queries to a given query engine.

- SPARQL queries have to be evaluated with a *query engine* to return results. Query engines are either available in software development kits, like Apache Jena,[76] RDF4J,[77] and RDFLib,[78] or integrated into triplestores.[79]
- *Validators* like the one offered by Sparql.org[80], on the one hand, provide a way to syntactically validate a SPARQL query and, on the other hand, annotate a query with SPARQL algebra primitives to debug the potential evaluation of a SPARQL query. This way, unexpected query results (e.g., due to the usage of group graph patterns and filters) can be caught, and optimization possibilities can be discovered.

As for *applications*, any application that somehow accesses a knowledge graph modelled with RDF is potentially an application of SPARQL. We will cover quite a bit about those applications in Part IV. Besides these, a typical application of SPARQL is the verification of RDF graphs via implementing constraints as SPARQL queries. In Sect. 13.2.2, we will cover a language that was born from this application of SPARQL, namely, SHACL.

[76] https://jena.apache.org/

[77] https://rdf4j.org/

[78] https://github.com/RDFLib/rdflib

[79] See Chap. 19 for details about triplestores.

[80] http://www.sparql.org/validator.html

13.2.2 SHACL

As briefly covered in Part I, the core technologies of the Semantic Web such as RDF, RDFS, and OWL have been developed following the open-world assumption. The open-world assumption allows incomplete knowledge. If a fact is not in an RDF graph, then it is unknown. This makes sense for the Web: who can guarantee that the knowledge on the Web is complete?

Contrarily, the closed-world assumption considers complete knowledge: which makes sense for several use cases, especially closed systems. Often, the RDF graphs described with RDFS/OWL ontologies are used in closed-world settings as applications may need data to fit certain constraints. However, OWL and RDFS are not for constraining an RDF graph but more for describing it.

Imagine an application that integrates data about local businesses in a city. The application may require that all instances of a local business must have at least one instance of opening hours specification. We can attempt to represent this constraint with description logic (and OWL, consequently):[81]

$$LocalBusiness \sqsubseteq \geq_1 hasOpeningHours.OpeningHoursSpecification$$

This description logic statement can be phrased as follows: "A LocalBusiness is something that has one or more opening hours specification values." Formally, however, this does not constrain the instances of local business but describes it. If we have a local business instance without any opening hours specification value, an OWL DL reasoner will not produce any errors, because of the open-world assumption. No opening hours specification in the RDF graph does not mean it does not exist. Until 2017, defining constraints over RDF graphs was done via SPARQL queries as SPARQL inherently works with the closed-world assumption. The SPARQL query approach, however, leads to the creation of ad hoc queries, which leads to low reusability, and there is always the risk that some SPARQL features may not be supported by all triplestores.

There were several proposals to define a more suitable approach to tackling this issue. For example, the language ShEx[82] used the notion of shapes to define constraints where an RDF graph or parts of it should fit. ShEx has an abstract syntax and formal semantics operationally defined based on graph structures. It is a popular non-standard community-based effort for constraining RDF graphs.

With the motivation of having a declarative language for defining constraints over RDF data, the Data Shapes Working Group of W3C has worked on several specifications. The *Shapes Constraint Language (SHACL)* was the resulting

[81] Do not worry if the representation is unfamiliar! We will cover OWL in Sect. 13.3 and Description Logic in Sect. 14.1.

[82] https://shex.io/shex-semantics/index.html

recommendation of this working group.[83] The goal of SHACL is to define "shapes" that a particular RDF graph (or a subgraph) should fit.

SHACL offers core constraint components that are more commonly needed based on the use cases collected during the development of the language.[84] There are also various extensions of the core language that allow the definition of more complex constraints. For example, SHACL-SPARQL is an extension that enables the definition of more complex constraints by using the full expressivity of SPARQL. This also allows the definition of custom constraint components, in which a parameterized SPARQL query is specified to tell SHACL verifiers how to check the custom constraints on an RDF graph. Although not part of the W3C recommendation, SHACL also offers so-called advanced features,[85] including SHACL Rules, where SPARQL queries can be used as inference rules.[86]

In the remainder of this section, we will introduce SHACL, particularly the modelling primitives in SHACL Core.[87] We will first explain some basic notions like shapes and focus and value nodes that are necessary for explaining all modelling primitives. Then, we cover different kinds of target declarations. Finally, we introduce different constraint components that can be used to verify RDF graphs. After the modelling primitives, we will introduce several tools and applications of SHACL. Finally, we will give a summary of the section.

We use the latest version of the SHACL specification; however, our aim is not to repeat the documentation. Instead, we offer straightforward descriptions of the modelling primitives and their typical usage[88,89]. SHACL constraints do not have formal semantics; however, at the implementation level, they are informally converted into SPARQL queries. To provide a better explanation of each modelling primitive, we use example shapes and their representation as SPARQL queries.

Note that the SPARQL implementations are informative and may have alternatives. In some cases, we will combine multiple constraints in one SPARQL query. This would increase the efficiency while trading off the granularity of the verification results.

[83] Section 13.2.2 contains content from https://www.w3.org/TR/shacl/. Copyright © 2017 W3C® (MIT, ERCIM, Keio, Beihang). W3C liability, trademark, and document use rules apply.

[84] https://www.w3.org/TR/shacl-ucr/

[85] https://www.w3.org/TR/shacl-af/

[86] https://www.w3.org/TR/shacl-af/#rules-examples

[87] https://www.w3.org/TR/shacl/#core-components

[88] The example data graphs are taken from the German Tourism Knowledge Graph: https://open-data-germany.org/en/open-data-germany/. The *dzt-entity* prefix is used for the namespace of the German Tourism Knowledge Graph.

[89] We benefited from Gayo et al. (2017) for structuring this section. Please refer there for a more comprehensive description of SHACL.

13.2.2.1 Basic Notions

We will cover basic notions like shape, focus nodes, and value nodes that are relevant for all modelling primitives.

Shape. The main modelling primitive of SHACL is a concept called shape**.** There are two kinds of shapes, namely, *node shapes* (e.g., an instance of sh:NodeShape) and *property shapes* (e.g., the value of sh:property in a shape). The shapes are stored in a *shapes graph*. The shapes graph is used to verify a data graph, an arbitrary RDF graph. The verification can be done via ASK or SELECT SPARQL queries. An RDF graph violates a shapes graph if the SELECT query of a constraint returns a result or the ASK query returns false.

Focus nodes. A node in a data graph that is validated against a shape using the triples from that data graph is called a focus node**.** For example, a shape instance can have target declarations that select a set of focus nodes for that shape.

Value nodes. For node shapes, the value nodes are the individual focus nodes. Note that depending on the target declaration (see below), more than one focus node can be selected; however, at any given time, only one focus node is active for a node shape. For property shapes with an *sh:path* value *p*, the value nodes are the set of nodes in the verified RDF graph that can be reached from the focus node via the path *p*.

13.2.2.2 Target Declarations

Target declarations select the focus nodes that will be verified against a shape. There are four different kinds of target declarations, node targets, class targets, subjects-of-targets, and objects-of-targets.

Node target. A node target selects the focus node identified with a given IRI. Figure 13.34 shows a data graph with two *schema:Event* instances.

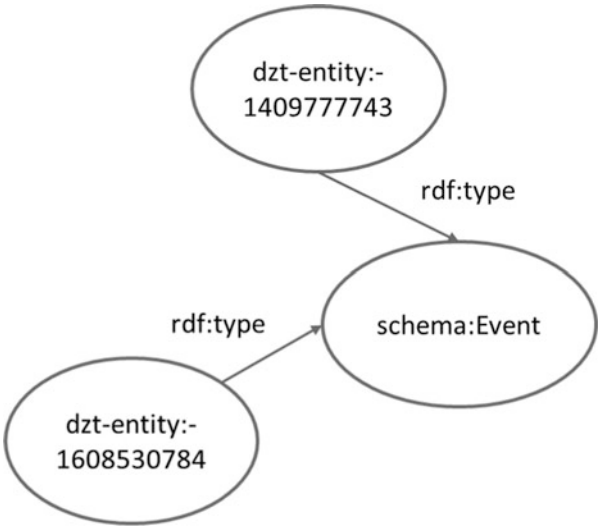


Fig. 13.34 A data graph with two Event instances

SHACL:

```
ex:SpecificEventShape a sh:NodeShape;
        sh:targetNode dzt-entity:1608530784.
```

SPARQL:

```
SELECT ?this where {
  BIND (dzt-entity:1608530784 as ?this) .
}
```

Fig. 13.35 A shape with a node target and a possible SPARQL implementation of the target declaration

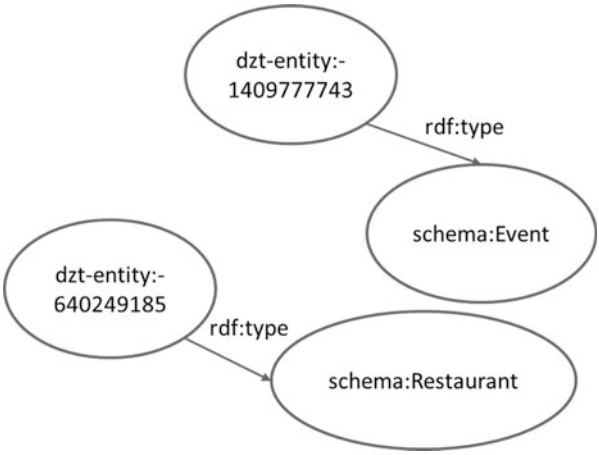


Fig. 13.36 A data graph with a schema:Restaurant and schema:Event instances

When the shape in Fig. 13.35 is implemented in SPARQL and run on the data graph, *dzt-entity:-1608530784* is returned as the focus node (bound to *?this*).

Class target. A class target selects focus nodes based on a given class IRI. All instances of the given class in the data graph are selected as focus nodes and validated individually. Figure 13.36 shows a data graph with two *schema:Event* instances.

When the shape in Fig. 13.37 is implemented as a SPARQL query and run on the data graph, *dzt-entity:-1409777743* is returned as the focus node (bound to *?this*). Note that the SPARQL implementation contains a property path *rdf:type/rdfs:subClassOf**.[90] This path allows the target declaration to select all instances of the given class, including the instances of all its subclasses. Such an implementation is necessary, because it is not guaranteed that the verified RDF graph supports RDFS entailment.[91]

[90] See more about property paths in SPARQL 1.1 Specification: https://www.w3.org/TR/sparql11-query/#propertypaths.

[91] See Sect. 14.2 for RDFS entailment.

```
SHACL:
ex:EventShape a sh:NodeShape;
        sh:targetClass schema:Event .
SPARQL:
SELECT DISTINCT ?this WHERE {
  ?this rdf:type/rdfs:subClassOf* schema:Event .
}
```

Fig. 13.37 A shape with a class target and a possible SPARQL implementation of the target declaration

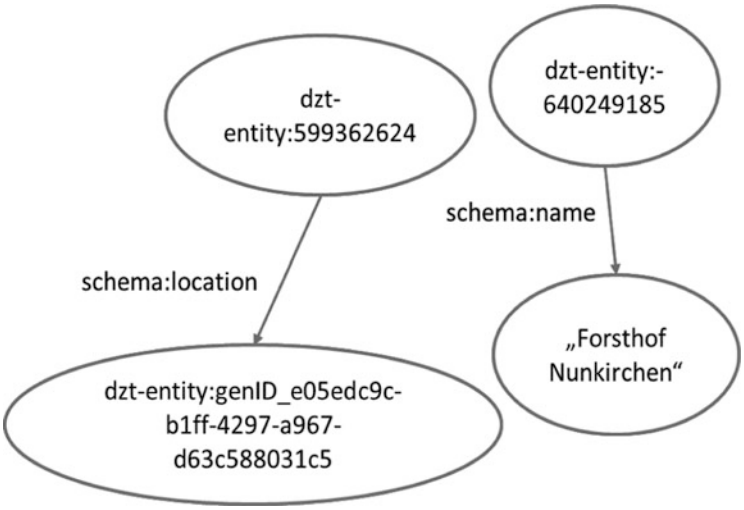


Fig. 13.38 A data graph with two triples with two different predicates

Subjects-of-targets. A subject-of-target selects focus nodes from an RDF graph that are in the subject position of a triple whose predicate is the given property. Figure 13.38 shows a data graph with two triples with *schema:location* and *schema:name* predicates.

When the shape in Fig. 13.39 is implemented as a SPARQL query and run on the data graph, *dzt-entity:599362624* is returned as the focus node since it is in the subject position for *schema:location* in a triple.

Objects-of-targets. An objects-of-target selects focus nodes in the object position of a triple from an RDF graph whose predicate is the given property. Let us take the data graph in Fig. 13.38 again.

When the shape in Fig. 13.40 is implemented as a SPARQL query and run on the data graph, *"Forsthof Nunkirchen"* is returned as the focus node since it is in the object position for *schema:name* in a triple.

SHACL:
```
ex:EventShape a sh:NodeShape;
        sh:targetSubjectsOf schema:location .
```

SPARQL:
```
SELECT DISTINCT ?this WHERE {
  ?this schema:location [] .
}
```

Fig. 13.39 A shape with a subjects-of-target and a possible SPARQL implementation of the target declaration

SHACL:
```
ex:NameShape a sh:NodeShape;
        sh:targetObjectsOf schema:name.
```

SPARQL:
```
select DISTINCT ?this where {
  [] schema:name ?this .
}
```

Fig. 13.40 A shape with an objects-of-target and a possible SPARQL implementation of the target declaration

13.2.2.3 Constraint Components

SHACL defines the notion of constraint components which are associated with shapes to declare constraints. Each shape can be associated with several constraint components. In the following sections, we will introduce cardinality constraints, value type constraints, value constraints, value range constraints, string-based constraints, logical constraints, shape-based constraints, and property pair constraints. Each constraint is first described informally, including its parameters, and then provided with a set of example shapes and SPARQL implementations for them on a given data graph. Note that we assume that the target declarations have already selected the focus nodes in our examples. Therefore, the SPARQL implementations of constraint components may be only given for a specific value node for the sake of conciseness. For the cases where there are multiple value nodes involved, the SPARQL query is parameterized. A parameter in a SPARQL query starts with the dollar sign *($)* followed by a string.

13.2.2.3.1 Cardinality Constraints

Cardinality constraints specify the minimum (*sh:minCount*) and/or maximum (*sh:maxCount*) number of values a given property can take in an RDF graph. A cardinality constraint can be only defined on properties.

- *sh:minCount* constrains the minimum number of values a property can have. A property with no *sh:minCount* constraint has no restriction on the minimum cardinality.
- *sh:maxCount* constrains the maximum number of values a property can have. A property with no *sh:maxCount* constraint has no upper bound for the number of values it can take.

Figure 13.41 shows a data graph with a *schema:Event* instance that contains two values for the *schema:startDate* property.

Figure 13.42 shows a shape that constrains the cardinality of the *schema:startDate* property as exactly 1 and its possible SPARQL implementation.[92] When run on the data graph in Fig. 13.41, the SPARQL query counts the *schema:startDate* values reachable from the focus node. The SPARQL query returns a result (violation) since the *schema:startDate* property has more than one value.

13.2.2.3.2 Value Type Constraints

Value type constraints specify what type a value node (e.g., the value of a given property) should have. This constraint component is typically used to enforce range constraints for properties. The following constraints can be defined:

- *sh:datatype*: values must be literals and have the given datatype.
- *sh:class*: values must be an instance of the given class.
- *sh:nodeKind*: restricts the kind of RDF term a value node should be. A value node can be a blank node, an IRI, a literal, or a pairwise disjunction of these types of terms.

Figure 13.43 shows a data graph with the *schema:location* property used on an instance. The property has a value which is a *schema:Place* instance.

Figure 13.44 shows a shape that constrains the range of the *schema:location* property to schema:PostalAddress and its possible SPARQL implementation. The property shape defines a path with *schema:location*; therefore, the value node is *dzt-entity:genID_e05edc9c-b1ff-4297-a967-d63c588031c5*. When run on the data graph in Fig. 13.43, the SPARQL query in Fig. 13.44 returns false (violation) since the value of the schema:location property is not an instance of schema:PostalAddress or any of its subclasses.

[92] Note that, typically, each constraint should be evaluated separately; however, we merge the implementation of minimum and maximum count constraints for the sake of conciseness.

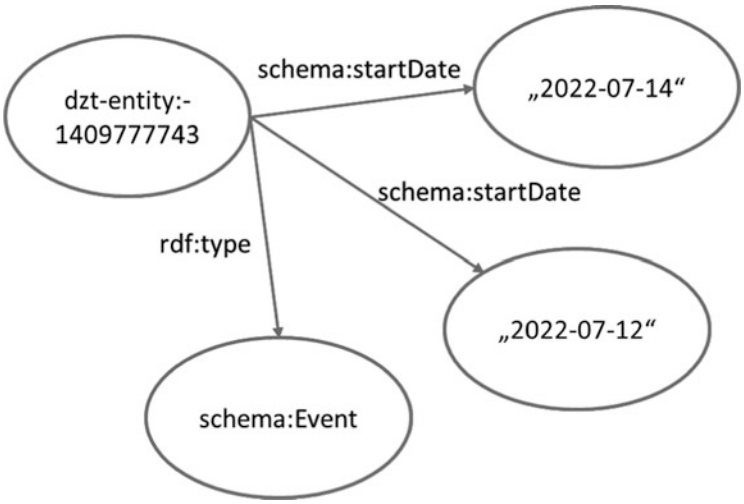


Fig. 13.41 A data graph with a schema:Event instance described with two properties

SHACL:

```
ex:EventShape a sh:NodeShape;
        sh:targetClass schema:Event;
        sh:property [
                sh:path schema:startDate;
                sh:minCount 1;
                sh:maxCount 1 .
        ]
```

SPARQL:

```
SELECT (COUNT(distinct ?startDate) as ?count) where {
  dzt-entity:-1409777743 schema:startDate ?startDate .
} HAVING (?count != 1)
```

Fig. 13.42 A shape with two cardinality constraints on the startDate property and its possible SPARQL implementation

The other two value type constraints can be implemented similarly on the same data graph shown in Fig. 13.43. Figure 13.45 shows a shape that constrains the range of the *schema:location* property to *xsd:string* and its possible SPARQL implementation. The SPARQL query returns false (violation) since the value of the *schema:location* property is not a valid literal and therefore does not have a datatype.

Finally, we take the same data graph in Fig. 13.43 to show the node kind constraints. Figure 13.46 shows a shape and its SPARQL implementation that constrains *schema:location* values so that they can be only IRIs. The SPARQL query returns true (no violation) since the value of the *schema:location* property is an IRI.

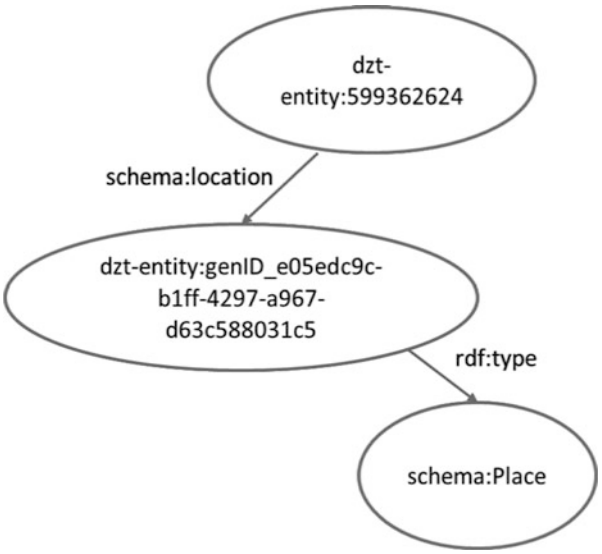


Fig. 13.43 A data graph with a location value that is a schema:Place instance

```
SHACL:
ex:EventShape a sh:NodeShape;
        sh:targetNode dzt-entity:599362624;
        sh:property [
                sh:path schema:location;
                sh:class schema:PostalAddress;
        ]

SPARQL:
ASK {
  dzt:genID_e05edc9c-b1ff-4297-a967-d63c588031c5
rdf:type/rdfs:subClassOf* schema:PostalAddress .
}
```

Fig. 13.44 A shape with a value type constraint (sh:class) on the location property and its possible SPARQL implementation

13.2.2.3.3 Value Constraints

Value constraints[93] specify restrictions on the values a property can have. A value constraint can be defined in two ways:

- *sh:hasValue*: specifies a value the property must have
- *sh:in*: enumerates the values that a property is allowed to have

[93] *sh:hasValue* and *sh:in* are not classified under any specific category in the SHACL specification; however, we classify them as *value constraints* for the sake of consistency across the section.

```
SHACL:
ex:EventShape a sh:NodeShape;
        sh:targetNode dzt-entity:599362624;
        sh:property [
                sh:path schema:location;
                sh:datatype xsd:string;
        ] .

SPARQL:
ASK {
  BIND (dzt:genID_e05edc9c-b1ff-4297-a967-d63c588031c5
as ?value) .
  FILTER (datatype(?value) = xsd:string)
}
```

Fig. 13.45 A shape with a value type constraint (sh:datatype) on the location property and its possible SPARQL implementation

```
SHACL:
ex:EventShape a sh:NodeShape;
        sh:targetNode dzt-entity:599362624;
        sh:property [
                sh:path schema:location;
                sh:nodeKind sh:IRI;]
SPARQL:
ASK {
  BIND (dzt:genID_e05edc9c-b1ff-4297-a967-d63c588031c5
as ?value) .
  FILTER (isIRI(?value))
}
```

Fig. 13.46 A shape with a value type constraint (sh:nodeKind) on the location property and its possible SPARQL implementation

Figure 13.47 shows a data graph with an instance whose address property has a blank node as a value. This blank node has *schema:addressRegion* property with the values "Bremen" and "Saarland."

Figure 13.48 shows a shape that constrains the value of the *schema: addressRegion* property, namely, that it should have at least one value as "Saarland," and its possible SPARQL implementation. When run on the data graph in Fig. 13.47, the SPARQL query returns true (no violation) since at least one value of the *schema: address/schema:addressRegion* property is "Saarland."

Similarly, Fig. 13.49 shows a shape that constrains the value of the *schema: addressRegion* property to a finite set of values, namely, "Hamburg" and "Bremen," and its possible SPARQL implementation. The SPARQL query returns a result (violation) since in the data graph in Fig. 13.47, there is a value for the property

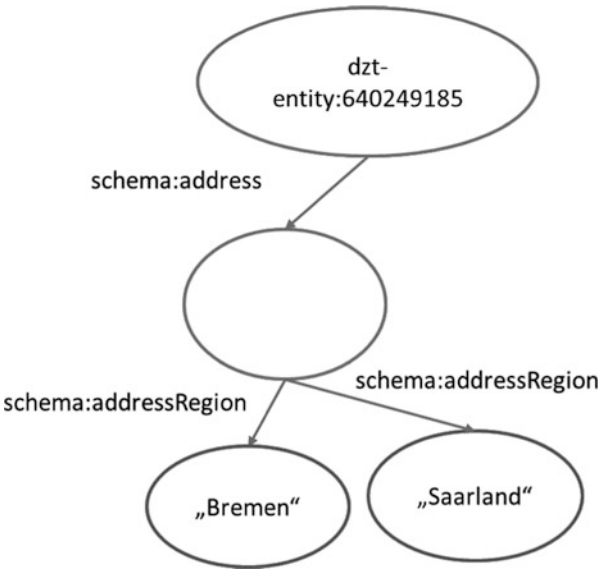


Fig. 13.47 A data graph that describes the region of a point of interest

SHACL:

```
ex:RegionShape a sh:NodeShape;
        sh:targetNode dzt-entity:640249185;
        sh:property [
                sh:path (schema:address schema:ad
        dressRegion);
                sh:hasValue „Saarland“;
        ]
```

SPARQL:

```
ASK {

  dzt-entity:640249185 schema:address/schema:addressRegion "Saar-
land" .

}
```

Fig. 13.48 A shape with a value constraint (sh:hasValue) and its possible SPARQL implementation

schema:addressRegion as “Saarland,” but it is not in the set of values specified by the constraint.

Note that the difference between *sh:hasValue* and *sh:in* may be subtle, but it is important. *sh:hasValue* only requires one property value to be the specified value (there can be other values for the same property), where *sh:in* allows only one of the specified values to be a value of the constrained property.

```
SHACL:
ex:RegionShape a sh:NodeShape;
        sh:targetNode dzt-entity:640249185;
        sh:property [
                sh:path (schema:address schema:addressRegion);
                sh:in („Hamburg“, „Saarland“);
        ]

SPARQL:
SELECT ?value
{
  BIND($region as ?value) # $region parameter is replaced with “Saarland” and “Bremen”
  FILTER (?value NOT IN („Hamburg", „Saarland"))
}
```

Fig. 13.49 A shape with a value constraint (sh:in) and its possible SPARQL implementation

13.2.2.3.4 Value Range Constraints

Value range constraints specify restrictions on the range of values. A value range constraint is defined via arithmetic comparison operators:

- *sh:minExclusive*: implemented as less-than operator (<)
- *sh:minInclusive*: implemented as less-than-or-equal operator (≤)
- *sh:maxExclusive*: implemented as greater-than operator (>)
- *sh:maxInclusive*: implemented as greater-than-or-equal operator (≥)

Figure 13.50 shows a data graph with a *schema:Rating* instance that has *schema:ratingValue* with the value 3.

Figure 13.51 shows a shape that constrains the value of the *schema:ratingValue* property to the range [1,5] and its possible SPARQL implementation.[94] When run on the data graph in Fig. 13.50, the SPARQL query returns true (no violation) since 3 is greater than or equal to 1 and less than or equal to 5.

13.2.2.3.5 String-Based Constraints

String-based constraints apply various restrictions on string representations of value nodes (e.g., values of a property). We cover four types of string-based constraints:

- *sh:minLength*: sets a lower bound to the length of a string value
- *sh:maxLength*: sets an upper bound to the length of a string value
- *sh:pattern*: checks whether the string values comply with a given regular expression

[94] Two constraints are merged in the implementation for the sake of conciseness.

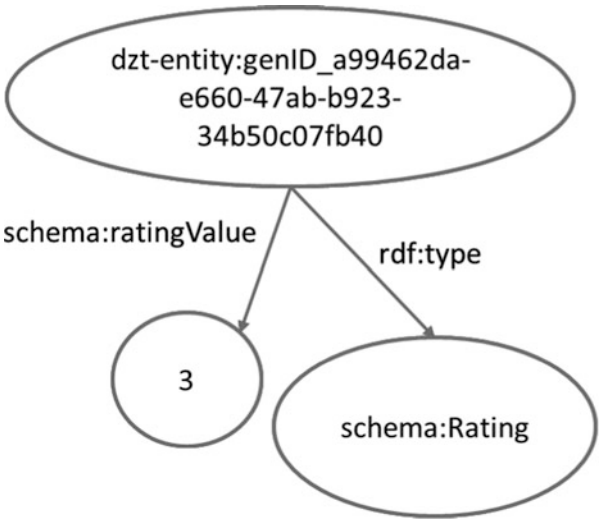


Fig. 13.50 A data graph with a schema:Rating instance and a value for schema:ratingValue property

SHACL:

```
ex:HotelRoomShape a sh:NodeShape;
        sh:targetClass schema:Rating;
        sh:property [
                sh:path schema:ratingValue;
                sh:minInclusive 1;
                sh:maxInclusive 5;
        ]
```

SPARQL:

```
ASK {
   BIND (3 as ?value).
   FILTER (1 <= ?value && 5 >= ?value)
}
```

Fig. 13.51 A shape with two value range constraints and its possible SPARQL implementation

- *sh:languageIn*: checks whether the string values are tagged with one of the given language tags

Note that the string-based constraints are not necessarily only used on string literals but on all values that can be converted to a string (e.g., IRIs) with the *STR* function of SPARQL. Figure 13.52 shows a data graph with one triple, where the telephone number of an instance is described.

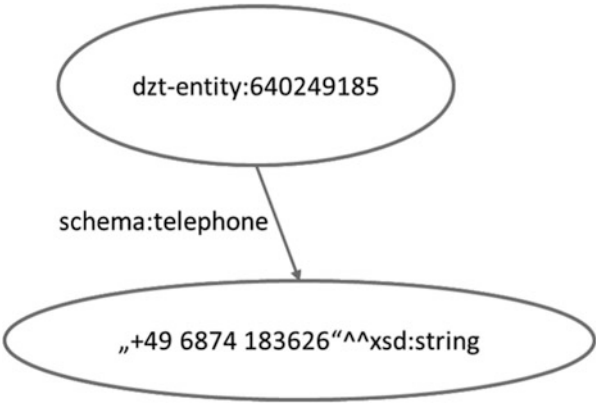


Fig. 13.52 A data graph with one triple that describes the telephone number of an instance

```
SHACL:
ex:TelephoneShape a sh:NodeShape;
        sh:targetNode dzt-entity:640249185;
        sh:property [
                sh:path schema:telephone;
                sh:minLength 1;
                sh:maxLength 15;
        ]

SPARQL:
ASK {
        BIND("+49 6874 183626" as ?value).
        FILTER (1 <= strlen(str(?value)) && 15 >= strlen(str(?value)))
}
```

Fig. 13.53 A shape with two string length constraints and its possible SPARQL implementation

Figure 13.53 shows a shape that constrains the length of the *schema:telephone* property value to the range [1,15] and its possible SPARQL implementation.[95] When run on the data graph in Fig. 13.52, the SPARQL query returns true (no violation) since the length of the telephone number is between 1 and 15 characters.

For the same data graph in Fig. 13.52, Fig. 13.54 shows a shape and its possible SPARQL implementation. The shape constrains the syntax of *schema:telephone* property value to be a given regular expression,[96] namely, that it should start with "+49." The SPARQL query returns true (no violation) since the telephone number fits the specified pattern. It starts with "+49."

[95] Two constraints are merged in the implementation for the sake of conciseness.

[96] In the XPath syntax: https://www.w3.org/TR/xpath-functions/#regex-syntax

```
SHACL:
ex:TelephoneShape a sh:NodeShape;
        sh:targetNode dzt-entity:640249185;
        sh:property [
                sh:path schema:telephone;
                sh:pattern „\\+49“
        ]

SPARQL:
ASK {
   BIND("+49 6874 183626" as ?value).
  FILTER (!isBlank(?value) && regex(str(?value), “\\+49”))
}
```

Fig. 13.54 A shape with a regular expression constraint on the schema:telephone property and its possible SPARQL implementation (Note that this is a simplified implementation in which RegEx configuration flags are not considered. See https://www.w3.org/TR/shacl/#PatternConstraintComponent for a potential implementation with flags)

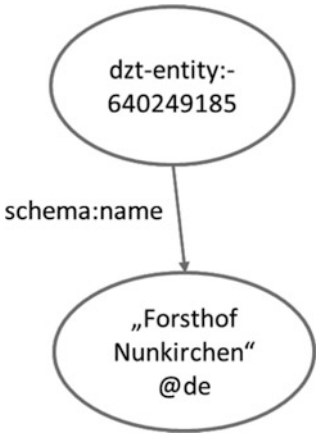


Fig. 13.55 A data graph with a triple describing the name of an instance

Let us look at another string-based constraint with the help of the data graph in Fig. 13.55. It contains a triple where the name of an instance is described.

Figure 13.56 shows a shape that constrains the allowed languages to German (de) for the values of *schema:name* and its possible SPARQL implementation. The SPARQL query returns true (no violation) since the literal “Forsthof Nunkirchen” is tagged with “de” for the German language.

13.2.2.3.6 Logical Constraints

Logical constraints connect a set of shapes with Boolean operators. Given a set of constraints, the following operators are supported:

```
SHACL:
ex:NameLangShape a sh:NodeShape;
        sh:targetNode dzt-entity:-640249185;
        sh:property [
                sh:path schema:name;
                sh:languageIn („de“)
        ]

SPARQL:
ASK {
  BIND("Forsthof Nunkirchen"@de as ?value) .
  FILTER (langMatches(lang(?value), "de"))
}
```

Fig. 13.56 A shape that constraints the language of the string value of schema:name and its possible SPARQL implementation

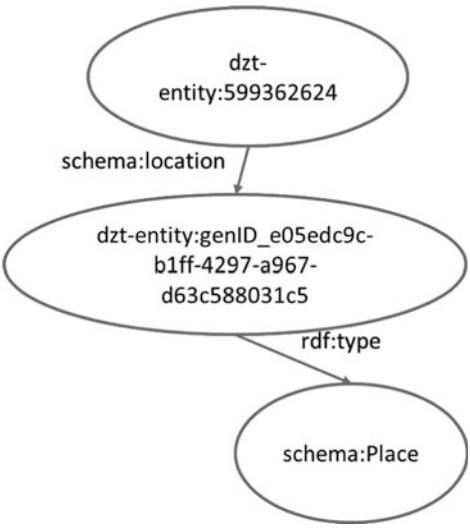


Fig. 13.57 A data graph with a triple where the schema:location property has a schema:Place instance as the value

- *sh:and*: all shapes must be satisfied by the value nodes.
- *sh:or*: at least one of the shapes must be satisfied by the value nodes.
- *sh:not*: the shape should not be satisfied by the value nodes.
- *sh:xone*: exactly one of the shapes must be satisfied by the value nodes.

Figure 13.57 shows a data graph where an instance has the *schema:location* property with a value that is an instance of *schema:Place.* The shape in Fig. 13.58 shows a constraint and its possible SPARQL implementation.[97] The constraint is not

[97] SHACL verifier implementations may verify all node shapes separately and not via a single query.

SHACL:

```
ex:EventShape a sh:NodeShape;
        sh:targetNode dzt:genID_e05edc9c-b1ff-4297-a967-d63c588031c5
;
        sh:property [
                sh:path schema:location;
                sh:or (
                        [
                          sh:class schema:PostalAddress;
                        ]
                        [
                          sh:class schema:Place;
                        ]
                    )
        ]
```

SPARQL:

```
ASK {
{
   dzt-entity:genID_e05edc9c-b1ff-4297-a967-d63c588031c5 rdf:type/rdfs:subClas-
sOf* schema:PostalAddress .
}
UNION
{
dzt-entity:genID_e05edc9c-b1ff-4297-a967-d63c588031c5 rdf:type/rdfs:subClas-
sOf* schema:Place .
}

}
```

Fig. 13.58 A shape with a logical constraint (sh:or) that puts two value type constraints in logical disjunction and its possible SPARQL implementation

violated when the value schema:location property is a *schema:PostalAddress OR* a *schema:Place* instance. When run on the data graph in Fig. 13.57, the SPARQL query returns true (no violation) because the value of *schema:location* property is a *schema:Place* instance.

13.2.2.3.7 Shape-Based Constraints

Shape-based constraints specify that the value node must conform to a given shape. In all examples until now, we have seen node shapes (e.g., instances of *sh:NodeShape*) and property shapes (e.g., values of *sh:property*) attached to them with certain constraints. A shape constraint can also be qualified. A qualified shape constraint enforces that a specified number of value nodes must satisfy a specific shape. This is typically used to define range constraints with specific requirements (e.g., a specific subtype of the range for several values). The following properties are supported:

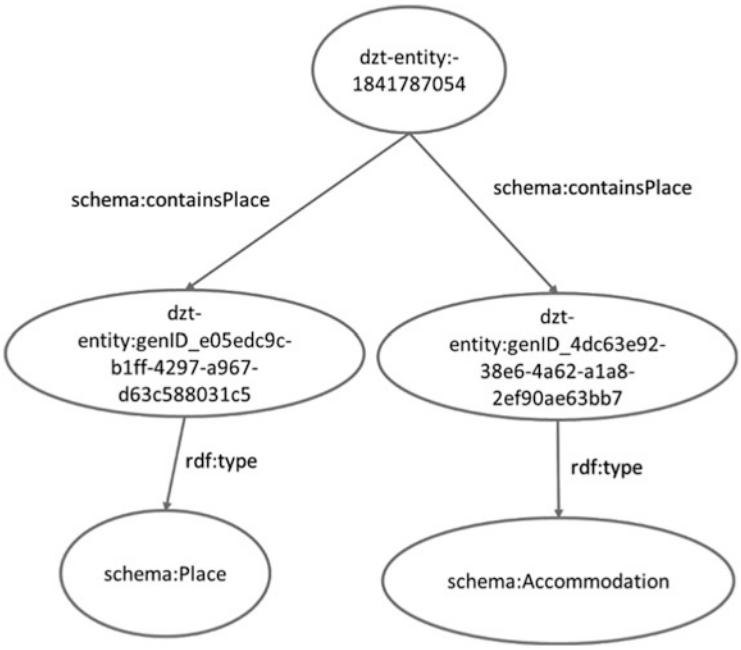


Fig. 13.59 A data graph containing an instance that has two schema:containsPlace property values, one schema:Place instance and one schema:Accommodation instance

- *sh:qualifiedValueShape*: the specific constraint which some of the values should satisfy
- *sh:qualifiedMinCount*: the minimum number of value nodes that should satisfy the specific constraint
- *sh:qualifiedMaxCount*: the maximum number of value nodes that should satisfy the specific constraint

Figure 13.59 shows a data graph with two values for *schema:containsPlace* property, one instance of *schema:Place* and another one that is an instance of *schema:Accommodation*.

Figure 13.60 shows a shape and its possible SPARQL implementation that constrains the range of the *schema:containsPlace* property on *schema:LodgingBusiness* instances. The property must have at least two values, and at least one must be an instance of *schema:Accommodation*. When run on the data graph in Fig. 13.59, the query does not return a value (no violation) since there are two values for *schema:containsPlace* and one of them is an instance of *schema:Accommodation*.

```
SHACL:
ex:LodginBusinessShape
        a sh:NodeShape ;
        sh:targetClass schema:LodgingBusiness ;
        sh:property [
                sh:path schema:containsPlace ;
                sh:minCount 2;
                sh:qualifiedValueShape [ sh:class ex:Accommodation ] ;
                sh:qualifiedMinCount 1 ;
        ].

SPARQL:

Select DISTINCT ?focus (COUNT(distinct ?value) as ?count)
(COUNT(distinct ?value2) as ?count2) where {
  BIND (<http://onlim.com/entity/-1841787054> AS ?focus)
  ?focus schema:containsPlace ?value .
  ?focus schema:containsPlace ?value2 .
  ?value2 rdf:type/rdfs:subClassOf* schema:Accommodation.

} GROUP BY ?focus HAVING (?count < 2 || ?count2 < 1)
```

Fig. 13.60 A shape that puts a qualified value shape on the property schema:containsPlace and its possible SPARQL implementation

13.2.2.3.8 Property Pair Constraints

Property pair constraints check whether the value nodes (e.g., values of a property) from a focus node (i.e., selected by a target declaration) have a specific relationship with the values of another property on the same focus node. These constraints can only be used in property shapes. The following properties are supported:

- *sh:equals*: the set of values of a given property must be equal to the set of values of the specified property on the focus node.
- *sh:disjoint*: the set of values of a given property must be disjoint from the set of values of the specified property on the focus node.
- *sh:lessThan*: every value of a given property must be less than all values of another property on the focus node.
- *sh:lessThanOrEquals*: every value of a given property at a focus node must be less than or equal to all values of the specified property on the focus node.

Figure 13.61 shows a data graph with two different ways of representing the telephone number of an instance.

Figure 13.62 shows a shape and its SPARQL implementation that checks whether the values of two property paths are equal. Note that the equality in SPARQL is syntactical. When the SPARQL query runs on the data graph in Fig. 13.61, the query does not return a value (no violation), since the set of values for *schema:telephone* is equal to the set of values for *schema:address/schema:telephone*. Note that the query would return a result (violation), if the number of elements in both sets was not

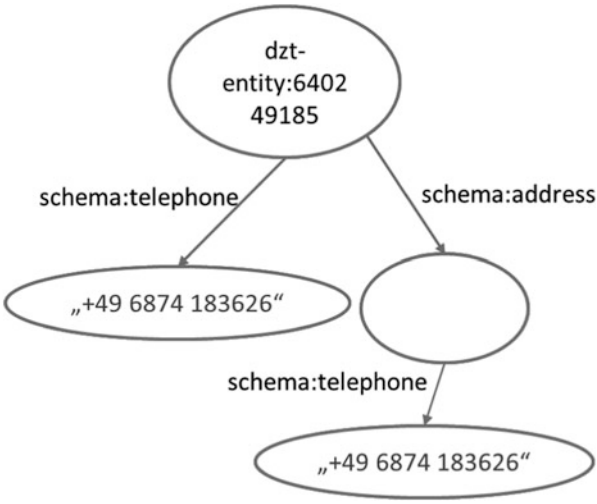


Fig. 13.61 A data graph that contains two values via different property paths for the schema:telephone property

```
SHACL:
ex:TelephoneShape a sh:NodeShape;
        sh:targetNode dzt-entity:640249185;
        sh:property [
                sh:path schema:telephone;
                sh:equals (schema:address schema:telephone)
        ]

SPARQL:
SELECT * WHERE{
  OPTIONAL{dzt-entity:640249185 schema:telephone ?value1.}
  OPTIONAL {dzt-entity:640249185 schema:address/schema:telephone ?value2.}
  FILTER((?value1 != ?value2) || (!bound(?value1) || !bound(?value2)))
}
```

Fig. 13.62 A shape that contains a property pair constraints (sh:equals) and its possible SPARQL implementation

equal. For that reason, the FILTER statement in the SPARQL implementation in Fig. 13.62 also checks if there is a result set where only one of the variables has a value.

Figure 13.63 shows a data graph with one instance described with the *schema: name* and *schema:alternateName* properties.

Figure 13.64 shows a shape and its possible SPARQL implementation that checks whether the set of values of *schema:name* and *schema:alternateName* are disjoint. When run on the data graph in Fig. 13.63, the SPARQL query returns a value (violation), since at least one value of *schema:name* equals to a value of *schema:alternateName*; therefore, they are not disjoint.

As for the final constraint, we look into the data graph in Fig. 13.65. It contains a *schema:Event* instance described with *schema:startDate* and *schema: endDate.*

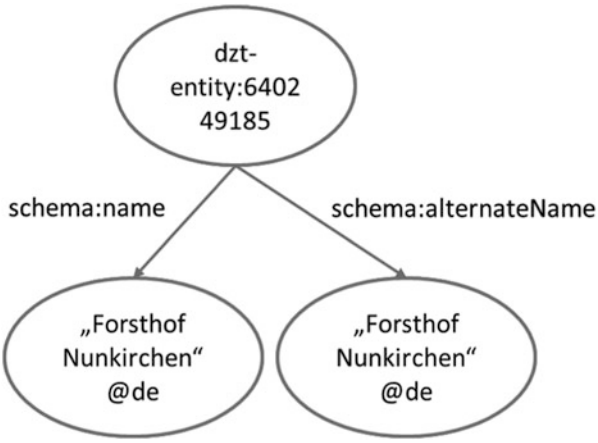


Fig. 13.63 A data graph where schema:name and schema:alternateName properties have the same value

SHACL:

```
ex:NameShape a sh:NodeShape;
        sh:targetNode dzt-entity:640249185;
        sh:property [
                sh:path schema:name;
                sh:disjoint schema:alternateName;        ]
```

SPARQL:

```
SELECT DISTINCT ?focus ?value WHERE {
        BIND (<http://onlim.com/entity/640249185> as ?focus)
        ?focus schema:name ?value .
        ?focus schema:alternateName ?value .
}
```

Fig. 13.64 A shape that contains a property pair constraint (sh:disjoint) and its possible SPARQL implementation

Figure 13.66 shows a shape and its SPARQL implementation that checks whether the value of *schema:startDate* is less than or equal to the value of *schema:endDate*. The SPARQL implementation of the shape returns a value, if two properties are not compatible (i.e., *!bound(?result)*) or the *?result* variable has the value false (i.e., *!(?result)*). When run on the data graph in Fig. 13.65, the query does not return a value (no violation) since both properties have a value and the start date is less than or equal to the end date.

So far, we have covered the most important SHACL Core constraints from an epistemological point of view. Note that there is no formal semantics for SHACL constraints; however, their semi-formal definitions are mostly implemented with SPARQL, even though this is not mandatory.

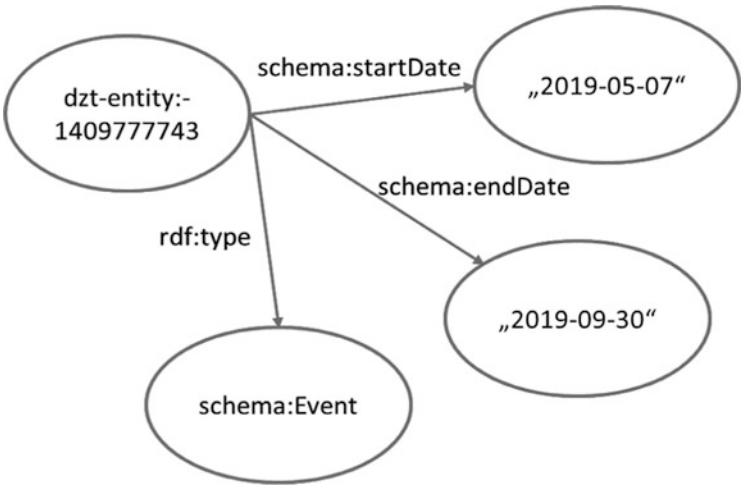


Fig. 13.65 A data graph where a schema:Event instance is described with its schema:startDate and schema:endDate properties

```
SHACL:
ex:StartEndDateShape a sh:NodeShape;
        sh:targetNode dzt-entity:-1409777743;
        sh:property [
                sh:path schema:startDate;
                sh:lessThanOrEquals schema:endDate;    ]


SPARQL:
SELECT ?focus ?value
WHERE {
        BIND (dzt-entity:1409777743> as ?focus)
        ?focus schema:startDate ?value .
        ?focus schema:endDate ?otherValue .
        BIND (?value <= ?otherValue AS ?result) .
        FILTER (!bound(?result) || !(?result)) .
}
```

Fig. 13.66 A shape that contains a property pair constraint (sh:lessThanOrEquals) and its possible SPARQL implementation

13.2.2.4 Tools and Applications

We will introduce some tools for creating and verifying SHACL constraints and applications that make use of SHACL.

The tools for SHACL can be categorized into two sets: *editors* and *verifiers*. As for editors, any text editor or a more advanced IDE[98] that supports RDF syntax can be used. We mentioned several of those already in Sect. 13.1. Below, we will mention several verifiers that can check SHACL constraints against RDF data.[99]

TopBraid SHACL API is a reference implementation of a SHACL verifier implemented as a Java library. It supports SHACL Core, SHACL-SPARQL, and SHACL Rules.[100]

SHACL Playground[101] is an online SHACL demo implemented in JavaScript where quickly shapes can be created and verified against small RDF datasets.

Corese STTL SHACL validator[102] is a SHACL Core implementation using STTL (SPARQL Template Transformation Language) (Corby et al. 2014), which is a generic transformation language for RDF.

RDFUnit[103] (Kontokostas et al. 2014) is a test-driven data quality framework that generates test cases for error detection in SHACL. It supports a major part of SHACL Core and SHACL-SPARQL.

VeriGraph[104] is a tool for verifying large knowledge graphs. It supports a subset of SHACL Core. It is also the underlying engine for the verification of semantic annotations based on domain specifications,[105] which are created as SHACL shapes.

There is a plethora of applications of SHACL, mainly targeting data quality tasks (e.g., error detection). A complete list of applications can be found in Gayo et al. (2017). Here, we mention the Springer Nature SciGraph and DBpedia as examples.

The *Springer Nature SciGraph*[106] is a Linked Open Data platform integrating various data sources, including Springer Nature and other partners from the scholarly publication domain. The platform contains knowledge from funders, research projects, conferences, organizations, and publications. SHACL is used to maintain a high data quality for the SciGraph (Hammond et al. 2017).

The *DBpedia Ontology*[107] has been maintained by the DBpedia community in a crowdsourced approach. To ensure a certain level of quality in the resulting ontology, it is verified with SHACL. For example, the ontology is checked whether every property has at most one domain and range definition or if properties and classes have at least one *rdfs:label* property assertion.

[98] https://en.wikipedia.org/wiki/Integrated_development_environment

[99] Mostly adapted from Gayo et al. (2017)

[100] https://github.com/TopQuadrant/shacl

[101] http://shacl.org/playground/

[102] http://corese.inria.fr/

[103] https://github.com/AKSW/RDFUnit/

[104] https://github.com/semantifyit/VeriGraph

[105] See Chap. 18.

[106] https://scigraph.springernature.com/explorer

[107] https://github.com/dbpedia/ontology-tracker

13.2.3 Summary

Data retrieval and manipulation are at the core of interacting with knowledge graphs, let it be by applications or humans. In this section, we introduced two languages for this purpose from an epistemological point of view. First, we introduced the SPARQL language, the standard language for querying and manipulating RDF data. The current 1.1 version of SPARQL supports various query types for retrieval such as SELECT, ASK, CONSTRUCT, and DESCRIBE as well as INSERT and DELETE queries for manipulation data in a knowledge graph. As a language for querying the RDF data model, SPARQL works based on matching triple patterns with the triples in an RDF graph.

Additional to querying, another way of manipulating RDF data is defining constraints on them. Due to the OWA of RDF, constraints were traditionally defined as ad hoc SPARQL queries since querying is inherently closed-world. SHACL is a rather recent language for defining constraints more declaratively, although their implementation is still typically done via SPARQL queries. Note that the implementation of SHACL with SPARQL is not necessarily normative; therefore, different implementations can take different approaches. SHACL offers many modelling primitives that allow the verification of nodes from an RDF graph against various constraints such as cardinality and range. It also allows the combination of different constraints via logical connectors.

In the next section, we will dive into the world of reasoning over data in a knowledge graph.

13.3 Reasoning over Data

A very powerful feature that comes with knowledge graphs is the implicit knowledge about a domain that can be made explicit via reasoning. For that, the domain that a knowledge graph is modelling must be described semantically, which means the facts and the terminology describing those facts must be defined.

In the following, we will introduce languages that can be used to describe the data in knowledge graphs semantically. We will first start with OWL and its successor, OWL2, the standard ontology languages for the Web. Then, we will talk about rules and how they can be used for knowledge representation on the Web. Finally, we wrap up with a summary of reasoning over data.

13.3.1 OWL and OWL2

For knowledge graphs, RDF(S) already provides a simple language to define factual and terminological knowledge. With RDF, objects can be instances of a class and

can have relationships with other objects. RDFS allows the provision of a hierarchy of classes and properties plus domain and range restrictions for properties.

However, compared to other logical languages, RDF(S) has quite limited expressivity:

- No all-quantification but only an enumeration of factual knowledge
- No cardinality restrictions, e.g., a person has minimum one name
- No disjointness of classes, e.g., to express that nothing can be a pizza and a wine at the same time

However, richer semantical means may help the Semantic Web to infer additional knowledge rather than collecting a large number of facts. Therefore, already with the beginning of the Semantic Web, there were immediate efforts to develop/adapt existing logical formalisms for the Web. First, *OIL* was developed as an ontology infrastructure for the Semantic Web. OIL is based on concepts developed in description logic and frame-based systems and is compatible with RDFS (Fensel et al. 2001). Meanwhile, *DAML-ONT* (McGuinness et al. 2003) was being developed as a parallel ontology language for the Semantic Web. In March 2001, the *Joint EU/US Committee on Agent Markup Languages* decided that DAML should be merged with OIL. The *EU/US ad hoc Joint Working Group on Agent Markup Languages* got together to further develop *DAML+OIL*[108] as a Web ontology language.

In 2001, a W3C Working Group started with the aim of defining a logical language for the Web. The OWL Web Ontology Language was developed as a family of several languages. Later, a second working group developed OWL2 based on the initial OWL specification. OWL and OWL2 are based on description logic (a subset of FOL) and developed to fulfill the following requirements (Antoniou et al. 2012):

- Well-defined syntax and semantics
- The convenience of expressions (epistemological level)
- Formal semantics
- Sufficient expressive power versus efficient reasoning support

In the following, we will introduce the modelling primitives of OWL and OWL2, respectively, with examples.[109] Finally, we will briefly mention tools for working with these languages and their applications.

[108] https://www.w3.org/TR/daml+oil-reference/

[109] For OWL examples, we mostly use the version 2.0 of the Pizza Ontology (https://protege.stanford.edu/ontologies/pizza/pizza.owl) by Alan Rector, Chris Wroe, Matthew Horridge, Nick Drummond, and Robert Stevens, distributed under CC-BY 3.0 license. See also Sack (2020) for a nice tutorial about OWL.

13.3.1.1 OWL

The first version of OWL was released in 2004 as a W3C recommendation. Since then, it has been a part of the Semantic Web Stack. It is more expressive than RDF (S). It can also be represented with the RDF data model and, consequently, its concrete syntaxes. Like RDFS, OWL uses the open-world assumption (OWA) and does not have the unique name assumption (UNA). This means we cannot state that something is false if it is not explicitly stated and two different names can refer to the same instance. This implies that OWL is inferencing additional facts and may introduce equality of syntactically different identifiers.

In the remainder of this section,[110] we will first introduce the basic notions of OWL that are necessary for understanding the content of this section and then its *dialects*, which are different subsets of the language with different expressivity. Finally, we dive into some common modelling primitives.

13.3.1.1.1 Basic Notions

Throughout this section, we will use certain notions to refer to different aspects of an OWL ontology. We will briefly introduce those in the following.[111]

Axioms. Axioms are statements made about a domain that are asserted to be true, for instance:

- Class axioms (:VegetarianPizza rdfs:subClassOf :Pizza)
- Property axioms *(:isToppingOf owl:inverseOf :hasTopping)*
- Individual axioms (assertions) *(:Germany owl:differentFrom :Italy)*

Entities. Entities are the basic elements of an ontology (identified by IRI), named terms, for example:

- Classes *(:Pizza—To characterize the set of all pizzas)*
- Properties *(:hasTopping—e.g., to represent the relationship between pizzas and their toppings)*
- Individuals *(:PetersPizza—To represent a particular pizza, an instance of the class :Pizza)*

Expressions. Expressions are complex definitions in the domain being described. For example, a combination of classes, properties, and individuals can form expressions. For instance, a class expression describes a set of individuals in terms of the restrictions on their characteristics. The following class expression defines the class *Spiciness* as a union of *Mild, Medium, and Hot.*

[110] The content of this section is mainly based on OWL specification (https://www.w3.org/TR/owl-ref), OWL2 syntax specification (https://www.w3.org/TR/owl2-syntax) and OWL2 primer (https://www.w3.org/TR/owl2-primer/). Copyright © 2004–2012 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[111] We use *:* as the empty prefix used for a default namespace in the examples.

```
pizza:Spiciness rdf:type owl:Class;
        owl:equivalentClass [rdf:type owl:Class;
                  owl:unionOf (pizza:Hot
                        pizza:Medium
                        pizza:Mild)];
```

13.3.1.1.2 OWL Dialects

There are OWL variations called dialects that realize a trade-off between expressivity and tractability. These variations are OWL Full, OWL DL, and OWL Lite (Fig. 13.67).[112]

OWL Full.[113] OWL Full allows the unrestricted use of RDF(S) and OWL constructs. It has very high expressivity but is also undecidable and therefore has no reasoning support. It natively allows unrestricted metamodelling, and therefore, it is beyond description logic.[114]

OWL DL.[115] OWL DL is a description logic which is a decidable fragment of FOL. It allows tractable reasoning while keeping a large expressiveness. It provides many modelling primitives of DL, such as negation, disjunction, full cardinality restrictions, enumerated concepts, and universal and existential quantification. It has, however, some limitations to achieve decidability and finite computation time, for example:

- Classes and individuals are disjoint.
- Data and object properties are disjoint.
- There are no cardinality constraints on transitively connected property chains.

OWL Lite.[116] OWL Lite is a restricted version of OWL DL. It mainly supports the following primitives; however, it is still a complex description logic:

- Class and property hierarchy
- Functional, inverse, transitive, and symmetric properties
- Individuals
- Conjunction of named classes
- (In)equality
- Existential and universal property restrictions
- Cardinality restrictions (only 0 or 1)

[112] https://www.w3.org/TR/owl-features/

[113] https://www.w3.org/TR/owl-ref/#OWLFull

[114] There are also DLs supporting meta logic, although with constraints. See for example (De Giacomo et al. 2009).

[115] https://www.w3.org/TR/owl-ref/#OWLDL

[116] https://www.w3.org/TR/owl-ref/#OWLLite

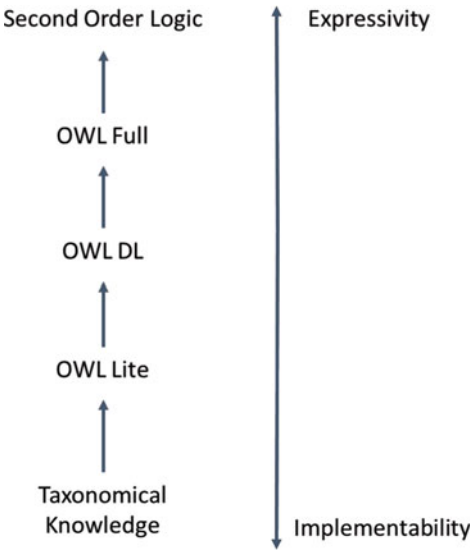


Fig. 13.67 OWL dialects and their relationship with other knowledge representation languages and logical formalisms (Figure adapted from Sack (2020))

13.3.1.1.3 Modelling Primitives

In the following, we will cover the modelling primitives of OWL 1.0 such as class expressions and axioms, property axioms, property restrictions, and individual axioms.

OWL class expressions and axioms. OWL has *owl:Class* definition and does not directly use the *rdfs:Class* type. Nevertheless, these two are aligned as follows:

```
owl:Class rdfs:subclassOf rdfs:Class
```

To be layered properly in the Semantic Web Technology Stack, OWL would be an extension of RDFS, in the sense that it adopts the RDFS meaning of *rdfs:Class* (a very powerful and expressive modelling primitive) and adds language primitives to support the richer expressiveness required. This approach would be consistent with the layered architecture of the Semantic Web. Unfortunately, simply extending RDF Schema would work against obtaining decidability and efficient reasoning (Antoniou et al. 2012) because not all RDFS classes are legal OWL DL classes. *owl:Class* and *rdfs:Class* are equivalent only in OWL Full.[117]

There are two classes that comprise the bottom and top of any class hierarchy in OWL. *owl:Thing* is the superclass of all classes in an ontology which contains all the individuals, whereas *owl:Nothing* is the empty class, which means it contains no individuals.

[117] https://www.w3.org/TR/owl-ref/#ClassDescription

```
pizza:VegetarianTopping rdf:type owl:Class ;
        owl:equivalentClass [ owl:intersectionOf ( pizza:PizzaTopping
                                                   [ rdf:type owl:Class ;
                                                   owl:unionOf ( pizza:CheeseTopping
                                                                 pizza:FruitTopping
                                                                 pizza:HerbSpiceTopping
                                                                 pizza:NutTopping
                                                                 pizza:SauceTopping
                                                                 pizza:VegetableTopping
                                                               )
                                                   ]
                                                   ) ;
                              rdf:type owl:Class
                            ] ;
```

Fig. 13.68 Example with owl:intersectionOf and owl:unionOf

OWL offers several primitives which are used to construct class expressions. These primitives mostly relate two or more classes in some way (e.g., conjunction, disjunction, equivalency, disjointness).

owl:intersectionOf describes a class that contains exactly those individuals that are members of both classes (i.e., conjunction). Given that each class represents a set of individuals, we describe the intersection of those sets with this primitive.

owl:unionOf describes a class that contains those individuals that belong to at least one of the given classes. Given that each class represents a set of individuals, we describe the union of those sets with this primitive (i.e., disjunction).

The class axiom in Fig. 13.68 specifies that *pizza:VegetarianTopping* is the class of all individuals that are in the intersection of *pizza:PizzaTopping* and a class that is the union of *pizza:CheeseTopping*, *pizza:FruitTopping*, *pizza:HerbSpiceTopping*, *pizza:NutTopping*, *pizza:SauceTopping*, and *pizza:VegetableTopping*.

owl:complementOf describes a class whose individuals do not belong to the given class. It is equivalent to a set complement.

The class axiom in Fig. 13.69 specifies that *pizza:NonVegetarianPizza* is the class of all *pizza:Pizza* individuals that are not *pizza:VegetarianPizza* individuals.

OWL also supports a pairwise disjointness predicate *owl:disjointWith* where two classes are declared disjoint. This indicates that the intersection of the set of individuals represented by these two classes is an empty set.

The axiom in Fig. 13.70 specifies that *pizza:Pizza* and *pizza:IceCream* are two disjoint classes. An individual can never be an instance of any two disjoint classes at the same time.

owl:equivalentClass specifies that two given classes have the same extension. That means any of the related classes have the same individuals. Figure 13.69 demonstrates the usage of *owl:equivalentClass* where *pizza:NonVegeterianPizza* is defined equivalent with the intersection of *pizza:Pizza* and the complement of *pizza:VegeterianPizza*. Note that the *owl:equivalentClass* primitive only indicates a set equality in terms of the members of two sets (i.e., extensions of two classes). So, two

```
pizza:NonVegetarianPizza rdf:type owl:Class ;
  owl:equivalentClass [ owl:intersectionOf ( pizza:Pizza
                        [ rdf:type owl:Class ;
                          owl:complementOf pizza:VegetarianPizza]);
                          rdf:type owl:Class
] ;
```

Fig. 13.69 An example with owl:complementOf

```
pizza:Pizza owl:disjointWith pizza:IceCream .
```

Fig. 13.70 An example with owl:disjointWith

equivalent classes do not necessarily refer to the same real-world concept; however, they have the exact same individuals.

OWL property axioms. OWL extends *rdf:Property* with two main disjoint types of properties, namely, object (owl:ObjectProperty) and datatype (owl: DatatypeProperty) properties. Object properties are expected to have individuals as values, while datatype properties are expected to only have literals as values.

Besides the primitives adopted from RDFS (e.g., *rdfs:domain, rdfs:range*), OWL provides several primitives that help to build property axioms, such as equivalent, functional, inverse functional, inverse, symmetric, and transitive property axioms.

owl:equivalentProperty specifies that two properties have the same extension.

The property axiom in Fig. 13.71 states that all property value assertions that hold for *pizza:hasIngredient* also hold for *pizza:containsIngredient* and vice versa. Note that, similar to the class equivalency, these two properties do not necessarily refer to the same real-world notion, but they have the exact same pairs in their extensions.

owl:FunctionalProperty specifies that given a functional property *P* and individuals *a*, *b*, and *c*, if *P(a,b)* and *P(a,c)* hold true, then *b* and *c* are the same individual.[118] This means a functional property can have only one distinct value for each instance on which it is used.

Similarly, *owl:InverseFunctionalProperty* specifies that given an inverse functional object property *P* and individuals, *a*, *b*, and *c*, if *P(a,c)* and P*(b,c)* hold true, then *a* and *b* are the same individuals.

The property axioms in Fig. 13.72 show that *pizza:isToppingOf* is a functional property; therefore, a specific instance of a topping cannot be used on two different pizzas. Similarly, *pizza:hasTopping* is defined as an inverse functional property; therefore, there cannot be two distinct pizzas that have the same instance of a topping.

[118] Remember that OWL adopts OWA and does not have UNA. If we do not explicitly state that *b* and *c* are different individuals, the ontology will still be consistent.

```
pizza:hasIngredient owl:equivalentProperty :containsIngredient .
```

Fig. 13.71 An example with owl:equivalentProperty

```
pizza:isToppingOf a owl:ObjectProperty ;
    a owl:FunctionalProperty .

pizza:hasTopping a owl:ObjectProperty;
    a owl:InverseFunctionalProperty .
```

Fig. 13.72 An example with owl:FunctionalProperty and owl:InverseFunctionalProperty

```
pizza:isToppingOf a owl:ObjectProperty ;
    owl:inverseOf pizza:hasTopping.
```

Fig. 13.73 An example with owl:inverseOf

owl:inverseOf specifies that given two inverse object properties *P* and *R* and individuals *a* and *b*, if *P(a,b)* is true, then *R(b,a)* is also true.

The property axiom in Fig. 13.73 shows that *pizza:isToppingOf* is an inverse property of *pizza:hasTopping*. For example, if a pizza has a salami instance as a topping, then that salami instance is a topping of that pizza.

owl:SymmetricProperty specifies that given a symmetric object property *P* and individuals *a* and *b*, if *P*(*a*,*b*) holds, then *P*(*b*,*a*) also holds.[119]

The property axiom in Fig. 13.74 states that if the food *a* pairs well with the food *b*, then *b* also pairs well with *a*.

owl:TransitiveProperty specifies that for an object property *P* and individuals *a*, *b*, and *c*, if *P*(*a*,*b*) and *P*(*b*,*c*) hold, then also *P*(*a*,*c*) holds.

The property axiom in Fig. 13.75 states that if *a* has the ingredient *b* and *b* has the ingredient *c*, then *a* has the ingredient *c*.

OWL property restrictions. OWL allows the definition of complex class expressions based on restrictions on their properties. *owl:Restriction* class allows definition of anonymous classes that have certain restrictions on specific properties. These restrictions can be mainly seen under two categories:

- Restrictions on property values (owl:hasValue, owl:allValuesFrom, owl: someValuesFrom)
- Restrictions on cardinality *(owl:cardinality, owl:minCardinality, owl: maxCardinality)*

[119] Careful readers may have already realized that another example for a symmetric property is *owl: inverseOf.*

```
:foodPairing a owl:ObjectProperty ;
      a owl:SymmetricProperty ;
      rdfs:domain pizza:Food ;
      rdfs:range pizza:Food .
```

Fig. 13.74 An example with owl:SymmetricProperty

```
pizza:hasIngredient a owl:ObjectProperty ;
      a owl:TransitiveProperty ;
      rdfs:domain pizza:Food ;
      rdfs:range pizza:Food .
```

Fig. 13.75 An example with owl:TransitiveProperty

```
pizza:ThinAndCrispyPizza rdf:type owl:Class ;
  owl:equivalentClass [ owl:intersectionOf ( pizza:Pizza
                        [ rdf:type owl:Restriction ;
                          owl:onProperty pizza:hasBase ;
                          owl:allValuesFrom pizza:ThinAndCrispyBase]) ;
                        rdf:type owl:Class
                      ] .
```

Fig. 13.76 An example with owl:allValuesFrom

We will briefly introduce universal (*owl:allValuesFrom*) and existential quantification (*owl:someValuesFrom*), as well as cardinality restrictions. Other primitives can be found in the OWL specification.[120]

owl:allValuesFrom defines an anonymous class that is the set of all individuals whose all values for the property *P* are members of the class *C*. Note that it does not cause any inconsistency, if an instance of that class does not have any value for the property *P*.

The expression in Fig. 13.76 defines that *pizza:ThinAndCrispyPizza* is a class of pizzas that have only *pizza:ThinAndCrispyBase* as base and vice versa. This means, if something is an instance of *pizza:ThinAndCrispyPizza*, its values of the *pizza:hasBase* property are all instances of *pizza:ThinAndCrispyBase*.

owl:someValuesFrom defines an anonymous class that is the set of all individuals where those individuals have at least one value for property *P* that is an instance of *C*.

The expression in Fig. 13.77 defines that *pizza:MeatyPizza* is a class of pizzas that have at least one value for their *pizza:hasTopping* property that is an instance of

[120] We will only consider object properties in our explanations; however, extension to these definitions to data properties is rather trivial.

```
pizza:MeatyPizza rdf:type owl:Class ;
   owl:equivalentClass [ owl:intersectionOf ( pizza:Pizza
                                [ rdf:type owl:Restriction ;
                                  owl:onProperty pizza:hasTopping ;
                                  owl:someValuesFrom pizza:MeatTopping]) ;
                               rdf:type owl:Class
                       ] ;
```

Fig. 13.77 An example with owl:someValuesFrom

```
pizza:InterestingPizza rdf:type owl:Class ;
   owl:equivalentClass [ owl:intersectionOf ( pizza:Pizza
                                [ rdf:type owl:Restriction ;
                                  owl:onProperty pizza:hasTopping ;
                                  owl:minCardinality 3]) ;
                               rdf:type owl:Class
                       ] ;
```

Fig. 13.78 An example with owl:minCardinality

pizza:MeatTopping. More informally, any pizza that has at least one meat topping is a meaty pizza. A meaty pizza can have additional non-meaty toppings.[121]

Finally, **cardinality restrictions** allow the definition of anonymous classes of all individuals that have a certain number of property values. The cardinalities can be restricted as minimum (*owl:minCardinality*), maximum (*owl:maxCardinality*), and exact (*owl:cardinality*) number of *distinct* values expected on a property.

The axiom in Fig. 13.78 defines *pizza:InterestingPizza* as a class of all pizzas that have minimum three values for the *owl:hasTopping* property. More informally, any pizza that has more than two *distinct* toppings is classified as an interesting pizza.

Individual axioms. OWL offers several primitives that relate two or more individuals to build axioms.

owl:sameAs states that two individuals are the same. This means they have the same property value assertions.

The axiom in Fig. 13.79 states that *:PetersPizza* is an instance of *pizza:VegetarianPizza* and *:JohnsPizza* is an instance of *pizza:MeatyPizza*. These two instances are the same individuals. However, *pizza:MeatyPizza* and *pizza:VegetarianPizza* are defined disjoint. This means *:PetersPizza* and *:JohnsPizza* cannot be the same individuals; therefore, the axioms in Fig. 13.79 are inconsistent.

owl:differentFrom defines an individual axiom that states two individuals are different. Figure 13.80 shows the statement that pizza:France and pizza:Germany are different individuals.

[121] Try creating a *pizza:MeatyPizza* instance with a vegetable topping and without any meat topping. Think about why this does not cause any inconsistencies.

```
:PetersPizza a pizza:VegetarianPizza.
:JohnsPizza a pizza:MeatyPizza.
:MeatyPizza owl:disjointWith :VegetarianPizza.
:PetersPizza owl:sameAs JohnsPizza.
```

Fig. 13.79 An example with owl:sameAs

```
pizza:France owl:differentFrom pizza:Germany.
```

Fig. 13.80 An example with owl:differentFrom

This primitive is particularly important due to the lack of the unique name assumption of OWL and RDF. For example, we can specify that two individuals are different explicitly, to cause inconsistencies from functional properties or cardinality restrictions. Assume that we have a restriction that a pizza is something that has only one country of origin. If we have a pizza with two countries of origins, namely, France and Germany, we can cause an inconsistency by explicitly specifying that France and Germany are different individuals, like in Fig. 13.80. Otherwise, it would only mean that France and Germany are the same individuals.

OWL has played a key role in an increasing number of applications in domains like eScience, e-commerce, geography, engineering, and defense. Still, the in-use experience with OWL led to the identification of restrictions on expressivity and scalability. A W3C OWL WG has updated OWL to fix certain limitations. The result is called OWL2 (Horrocks 2009).

13.3.1.2 OWL2

OWL2[122] was released in 2012 as the new version of the OWL language. It is fully backward compatible with OWL. Therefore, every OWL ontology is a valid OWL2 ontology, and every OWL2 ontology not using new features is a valid OWL ontology. It extends OWL with a small but useful set of core features:

- Syntactic sugar for some OWL1.0 constructs (e.g., disjoint union of classes, multiple disjoint classes, negative property assertions)
- Extended annotation capabilities (e.g., annotation of complex axioms, ontologies)
- Adds profiles: new language subsets with useful computational properties
- New features for more expressivity:
  - Qualified cardinality restrictions
  - Asymmetric, reflexive, and disjoint properties

[122] https://www.w3.org/TR/owl2-overview/

- Property chains
- Keys
- Simple metamodelling (e.g., punning)
- Restricted datatype features (e.g., custom datatype definitions, data ranges)

In the remainder of this section, we first introduce the new OWL2 profiles that are variations of OWL DL optimized for different tasks and purposes. Then, we describe the new features.[123]

13.3.1.2.1 OWL2 Profiles

Many years of usage of OWL in different domains for different tasks made some requirements clearer. For example, in life sciences domain, practitioners typically deal with large terminological knowledge, and they are willing to trade off some expressivity with a more favorable complexity for certain reasoning tasks. Similarly, applications that work with databases typically have large assertional knowledge, and they focus more on querying than other reasoning tasks.

Based on these observations, OWL2 introduces three subsets of OWL DL that are optimized for different tasks and purposes. These profiles are *OWL EL*, *OWL RL*, and *OWL QL*. Figure 13.81 shows the relationship between the new profiles and OWL dialects. In the following, we will briefly introduce the new profiles and their main features and motivation.

OWL EL[124] is a profile that provides polynomial time reasoning for TBox and ABox. It is particularly useful for reasoning with ontologies with large terminological part. It has the following main features:

- Existential quantification to a class expression
- Existential quantification to an individual
- Class intersections
- A subset of OWL DL axiom constructs
- Class inclusions, equivalence, disjointness
- Property inclusion, equivalence; transitive and reflexive properties

OWL RL[125] is a profile designed to accommodate scalable reasoning. It trades expressivity for efficiency. It is particularly meant for RDFS applications that need some extra expressivity. It has syntactic restrictions on Description Logic. Consequently, it does not support certain OWL2 DL modelling primitives, such as disjoint unions. *It also provides a link to the rule-based reasoning* style which should have been the task of OWL Lite.

[123] We refer the readers to https://www.w3.org/TR/owl2-new-features for a detailed description of new features in OWL2.

[124] https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/#OWL_2_EL

[125] https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/#OWL_2_RL

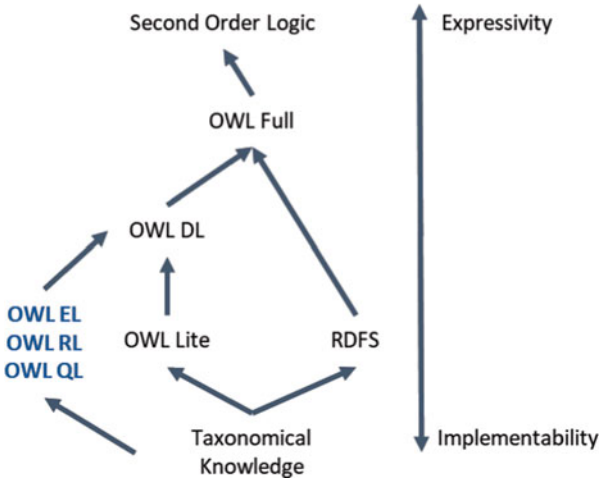


Fig. 13.81 The relationship between the new profiles and OWL dialects [Figure adapted from Sack (2020)]

OWL QL[126] is designed for sound and complete reasoning in LOGSPACE with respect to the size of the ABox of a knowledge base. Therefore, it is useful for OWL knowledge bases with a large ABox. Like OWL RL, it has some syntactic restrictions and restrictions on the supported modelling primitives. For example, transitive, functional, and inverse-functional properties are not supported by OWL QL. Equality of individuals (*owl:sameAs*) is not directly supported but can be handled with some preprocessing.

13.3.1.2.2 Modelling Primitives

In the following, we will briefly introduce some important new modelling primitives brought by OWL2 with examples. These are, namely, qualified cardinality restrictions, new property axioms primitives, disjoint properties, property chains, punning, and restricted datatypes.

Qualified cardinality restrictions. OWL2 supports class constructors with number restrictions on properties connected with a range constraint. This restriction specifies the minimum (*owl:minQualifiedCardinality*), maximum (*owl:maxQualifiedCardinality*), or exact (*owl:qualifiedCardinality*) qualified cardinality restrictions.

The axiom in Fig. 13.82 extends the *pizza:InterestingPizza* definition (Fig. 13.78) with a qualified cardinality restriction. In this example, *pizza:InterestingPizza* is defined as a *pizza:Pizza* instance that has at least three distinct *pizza:PizzaTopping* instances as value for the *pizza:hasTopping* property.

[126] https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/#OWL_2_QL

```
pizza:InterestingPizza rdf:type owl:Class ;
   owl:equivalentClass [ owl:intersectionOf ( pizza:Pizza
                                [ rdf:type owl:Restriction ;
                                  owl:onProperty pizza:hasTopping ;
                                  owl:minQualifiedCardinality 3;
                                  onClass pizza:PizzaTopping]) ;
                               rdf:type owl:Class
                       ] ;
```

Fig. 13.82 An example with owl:minQualifiedCardinality

```
:isFatherOf a :AsymmetricProperty ;
            rdfs:range :Person ;
            rdfs:domain :Person .
```

Fig. 13.83 An example with owl:AsymmetricProperty

New property axiom primitives. OWL2 extends the property constructors provided by the first version with asymmetric, reflexive, and irreflexive properties.

owl:AsymmetricProperty specifies that an object property is asymmetric. This means that given the property *P* and individuals a and b, if *P*(*a,b*) holds, then *P*(*b*,*a*) cannot hold.

The axiom in Fig. 13.83 states that if a person *A* is the father of person *B*, then *B* cannot be the father of person *A*.

owl:ReflexiveProperty specifies that an object property is reflexive, which means it connects an individual with itself.

The axiom in Fig. 13.84 states that given a reflexive property *:partOf*, for every individual *a* in a knowledge base, *:partOf(a,a)* holds. More informally, everything is a part of itself.

Note that *owl:ReflexiveProperty* defines reflexivity of a property globally. OWL2 also provides a mechanism for defining locally reflexive properties that are only reflexive for the instances of a certain type.

The axiom in Fig. 13.85 states that every *:Narcist* instance *a* is a *:Person* instance for which *loves(a,a)* holds true.[127] More informally, narcists love themselves.

owl:IrreflexiveProperty specifies that an object property is irreflexive, which means that it cannot connect an individual with itself.

The axiom in Fig. 13.86 states that given an irreflexive property *:properPartOf*, for any individual *a* in a knowledge base, there cannot be any property assertion : *properPartOf(a,a).* More informally, nothing can be a proper part of itself.

[127] Note that this time we have subclass relationship instead of equivalent class, which means not everybody who loves themselves are narcists.

```
:partOf a owl:ReflexiveProperty .
```

Fig. 13.84 An example with owl:ReflexiveProperty (Example adapted from https://www.w3.org/TR/owl2-new-features/#F6:__Reflexive.2C_Irreflexive.2C_and_Asymmetric_Object_Properties)

```
:Narcist rdf:type owl:Class ;
        rdfs:subClassOf [ owl:intersectionOf ( :Person
                                               [ rdf:type owl:Restriction ;
                                                 owl:onProperty :loves ;
                                                 owl:hasSelf "true"^^xsd:boolean
                                               ]
                                             ) ;
                          rdf:type owl:Class
                        ] .
```

Fig. 13.85 An example with owl:hasSelf

```
:properPartOf a :IrreflexiveProperty .
```

Fig. 13.86 An example with owl:IrreflexiveProperty (Example adapted from https://www.w3.org/TR/owl2-new-features/#F6:__Reflexive.2C_Irreflexive.2C_and_Asymmetric_Object_Properties)

Disjoint properties. OWL2 supports not only class disjointness but also property disjointness for two (*owl:propertyDisjointWith*) or more (*owl:AllDisjointProperties*) properties.

The axiom in Fig. 13.87 states that given two disjoint properties *:hasUncle* and *:hasAunt*, there cannot be two individuals *a* and *b*, for which *:hasUncle(a, b)* and *:hasAunt(a, b)* hold at the same time. More informally, someone cannot be an uncle and aunt of someone at the same time.

Property chains. RDFS provides the *rdfs:subPropertyOf* primitive which allows us to create property hierarchies. This primitive is adopted by OWL and enriched by OWL2 with the capability of constructing property chain inclusions. Figure 13.88 specifies that *:hasUncle* property is composed from a chain of *:hasFather* and *:hasBrother* properties. This effectively defines a property chain *:hasFather* and *:hasBrother* that is a subproperty of *:hasUncle.*

The axiom in Fig. 13.88 states that given the individuals a, b, and c, if *:hasFather (a, b)* and *:hasBrother(b, c)* hold, then *:hasUncle(a, c)* also holds.

Keys. OWL2 provides a primitive (*owl:hasKey*) to state that a set of properties uniquely identify an instance. If two individuals of a class have the same values for

```
:hasUncle a owl:ObjectProperty ;
        owl:propertyDisjointWith :hasAunt .
```

Fig. 13.87 An example with owl:propertyDisjointWith

```
:hasUncle owl:propertyChainAxiom ( :hasFather :hasBrother ) .
```

Fig. 13.88 An example with owl:propertyDisjointWith

```
:Person a owl:Class ;
        owl:hasKey ( :hasSsn ).
```

Fig. 13.89 An example with owl:hasKey

the properties specified by *owl:hasKey* for that class, then they are the same individuals.

The axiom in Fig. 13.89 states that two *:Person* instances that have the same *:hasSsn* property value are the same individuals.

Punning. In knowledge modelling, treating individuals as classes is called metamodelling. OWL 2.0 supports metaclasses through punning: The same subject is interpreted as a class and an individual. The decision is made by the context. Even when the names are identical, the underlying logic will treat them as if they have different names.

Restricted datatypes. OWL2 allows the definition of customized datatypes based on a wide range of existing XSD datatypes.

The example in Fig. 13.90 defines a new datatype called *:ssnDataType*[128] that contains all 11-character-length strings. This datatype can be used, for example, as the range of the property *:hasSsn*.

OWL2 extends OWL with three profiles to allow for use case-specific optimizations to enable for higher performance, straightforward implementation, and scalability to larger amounts of data. It also introduced several new modelling primitives that overcome many of the limitations of OWL 1.0. OWL2 is currently supported by many well-established reasoners and superseded OWL 1.0 in terms of usage.

13.3.1.3 Tools and Applications

We provide a brief overview of tools that work with OWL ontologies and application areas where OWL ontologies are used.

[128] Social Security number.

```
:ssnDataType a rdfs:Datatype ;
        owl:onDatatype xsd:string ;
        owl:withRestrictions (
          [ xsd:minLength "11"^^xsd:integer ],
          [ xsd:maxLength "11"^^xsd:integer ]) .
```

Fig. 13.90 An example with datatype restrictions

We categorize tools for OWL under editors, software libraries, and reasoners. Obviously, it will not be possible to cover all the tools here; however, we can give some examples. A larger list of tools can be found in online compilations.[129]

- *Ontology editors* are typically graphical tools that help knowledge engineers to model ontologies in OWL. In recent years,[130] there has been a significant increase in the number of ontology editors, but still the most adopted tool is undoubtedly Protégé.[131] Protégé supports OWL2 ontologies and provides a graphical interface for building class and property hierarchies, defining class expressions and more. It also has built-in reasoner support, and its functionality can be extended via plugins.
- To work with ontologies in a programming environment, typically, a *software development kit (SDK)* is needed. A very popular one is Jena,[132] which has been maintained for many years and contains an Ontology API for dealing with OWL and RDFS ontologies.
- Finally, *reasoners* are a crucial piece of software to make use of OWL ontologies. Their main purpose is to make implicit knowledge defined in an ontology explicit and find inconsistencies. Popular reasoners for OWL DL are Hoolet,[133] FaCT++,[134] and Pellet.[135]

Enumerating applications of OWL would not be the most efficient use of space in a textbook, given how extensively it is used to develop ontologies on the Web. OWL ontologies have been developed for various purposes, in a variety of domains such as education, health, life sciences, manufacturing, materials, smart buildings, and many more.

Dumontier (2012) compiles a list of applications of OWL ontologies, primarily in the health and life sciences domains. As an example, SNOMED CT[136] is a widely used ontology that contains classes, properties, and their axioms about clinical terms.

[129] For example, https://github.com/semantalytics/awesome-semantic-web#ontology-development

[130] For example, http://owlgred.lumii.lv/get_started and https://cognitum.eu/semantics/fluenteditor/

[131] http://protege.stanford.edu/

[132] https://jena.apache.org/documentation/ontology/

[133] http://owl.man.ac.uk/hoolet/

[134] http://owl.man.ac.uk/factplusplus/

[135] http://clarkparsia.com/pellet/

[136] https://www.snomed.org/snomed-ct/why-snomed-ct

It offers the most comprehensive medical terminology with approximately 400K classes. From an expressivity point of view, it is in OWL EL. It is currently used in healthcare systems of dozens of countries for knowledge interoperability and powering decision support systems.

A recently published ontology catalog[137] lists dozens of ontologies in industrial domains such as materials and manufacturing. Large organizations use OWL ontologies for explicit representation of internal knowledge and knowledge interoperability across different stakeholders working together in the same domain.

There are naturally many other academic and industrial applications. We refer the readers to repositories like Linked Open Vocabularies[138] for a detailed look.

13.3.2 Rules

Knowledge-based systems can be typically built with two different kinds of rules, production and logic rules.

On the one hand, production rules work with the condition-action principle, which means they are typically in the form of IF *premise(s)* THEN *action.* Production rules are typically state-based and procedural. At a given state of computation, an applicable rule (their premises given the current state are evaluated to true) is selected and updates the state. Obviously, different selection strategies lead to different results of the computation.

On the other hand, logic rules are supposed to be declarative. The order the rules are given does not matter, and a reasoning engine derives new facts from them.[139] The answer can be the answer to a query or the complete set of derivable facts.

In this section, we first cover the connection between different kinds of rules and the Web via an interchange format called the *Rule Interchange Format (RIF)*,[140] and then we introduce two rule-based formalisms for knowledge representation that also influenced the RIF work, namely:

- The simple language Datalog (Ceri et al. 1989)[141]
- Its extension by F-logic[142] (Kifer 2005; Kifer and Lausen 1989)

Finally, we present some tools and applications.

[137] https://data.ontocommons.linkeddata.es/index

[138] https://lov.linkeddata.es

[139] There are also logic programming languages that have procedural features, such as Prolog with its cut operator.

[140] https://www.w3.org/TR/rif-overview/

[141] https://en.wikipedia.org/wiki/Datalog

[142] https://en.wikipedia.org/wiki/F-logic

13.3.2.1 Rule Interchange Format (RIF)

Along with the languages like RDFS and OWL, there was an increasing interest in using rules as a way of representing knowledge on the Semantic Web. The main challenge was, however, the landscape of rule languages was quite heterogeneous. The heterogeneity came from both technical and commercial perspectives. On the one hand, there were many different paradigms and syntaxes (e.g., standard first-order logic, logic programming, production rules); on the other hand, there are varying commercial interests and perspectives of people from academia and industry. Therefore, an interchange format called *Rule Interchange Format (RIF)* was developed as a W3C recommendation.[143]

RIF provides a framework, i.e., a set of intersecting dialects that cover different paradigms for rules. This way, rules from one language can be mapped to RIF as an intermediate format and then mapped to another rule language while preserving its semantics. Figure 13.91 depicts a Venn diagram[144] showing how the different dialects are related.

The dialects are briefly defined by the RIF specification as follows (see also Kifer (2011) and Straccia (2013)):

- *RIF-FLD*[145] is the Framework for Logic Dialects. RIF-FLD is not a dialect by itself, but it provides the syntactical and semantic framework for defining other logical dialects. It contains primitives from various logical formalisms (e.g., first-order logic) and supports different semantics (e.g., standard FOL, minimal model semantics). Other logic dialects are defined as a subset of this framework, which improves the extensibility of RIF.
- *RIF-BLD*[146] is the Basic Logic Dialect. This dialect corresponds to Horn logic[147] with equality and standard FOL semantics with different syntactic and semantic extensions. For example, a noteworthy syntactic extension is the support for the frame syntax of F-logic. The semantic extension, for instance, includes the support for OWL and RDF constructs. RIF-BLD is developed as a starting point for logic rules, and it can be extended within RIF-FLD.
- *RIF-PRD*[148] is the Production Rule Dialect. It is the other major dialect developed by the RIF Working Group alongside RIF-BLD. Production rules are not based on any logic and do not have declarative semantics. Therefore, RIF-PRD is not a

[143] The content of this section is mainly created based on RIF specification. Copyright © 2013 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply. https://www.w3.org/TR/rif-overview/

[144] Figure adapted from https://www.w3.org/TR/rif-ucr/Copyright © 2013 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

[145] http://www.w3.org/TR/2013/REC-rif-fld-20130205/

[146] https://www.w3.org/TR/2013/REC-rif-bld-20130205/

[147] See Chap. 2 and Sect. 14.1.4 for an explanation for Horn logic.

[148] http://www.w3.org/TR/rif-prd

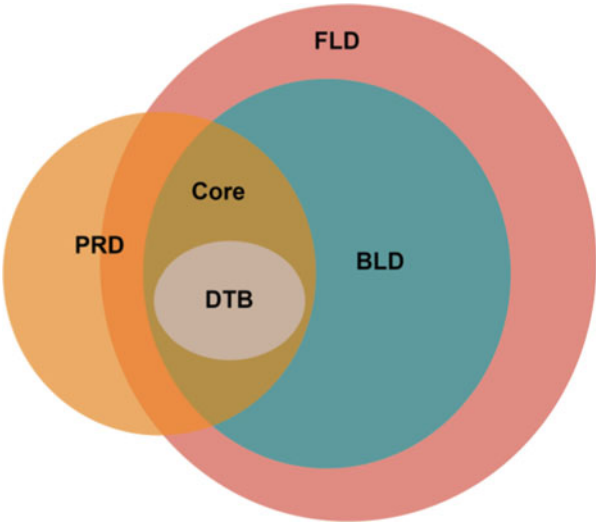


Fig. 13.91 Venn diagram of RIF dialects

logic-based RIF dialect. Still, an intersection of RIF-PRD and RIF-BLD is defined, which led to the development of RIF-Core.

- *RIF-Core*[149] is the Core Dialect. This dialect is an intersection of RIF-BLD and RIF-PRD dialects. It enables rule exchange between those two dialects to some extent. RIF-Core is basically Horn logic without function symbols (i.e., Datalog) with various extensions to support features such as objects and frames as in F-logic, internationalized resource identifiers for concepts, and XML datatypes and built-in functions (a part of RIF-DTB).
- *RIF-DTB*[150] contains XML datatypes and built-in functions that help bridge the gap between rules and the (Semantic) Web. Rules often use built-in functions and operators (e.g., arithmetic operators, string manipulation functions) and datatypes (e.g., integers, strings, Booleans). RIF-DTB contains well-defined and identified datatypes and built-ins based on XML specification to foster the exchange of rules on the Web.

Syntactically and semantically, RIF is compatible with the existing Semantic Web Stack technologies such as RDF(S) and OWL, which makes it a suitable format for the Web.[151] For example, resources in an RDF graph can be used in the rules as facts, and predicate symbols are also identified with IRIs. A knowledge base can be implemented with a combination of OWL and rules with the RIF syntax.

[149] http://www.w3.org/TR/rif-core

[150] https://www.w3.org/TR/2013/REC-rif-dtb-20130205/

[151] See the compatibility of RIF with RDF and OWL in the specification https://www.w3.org/TR/2013/REC-rif-rdf-owl-20130205/.

The Core and Basic Logical Dialect of RIF have been heavily influenced by Datalog and F-logic. In the following, we will introduce the modelling primitives of these two languages.

13.3.2.2 Datalog

Datalog (Ceri et al. 1989) has been developed as a language for deductive databases based on relational databases. Pure Datalog is a subset of Prolog without negation and function symbols, which means it is decidable. The decidability ensures that every Datalog query returns an answer. The language is also fully declarative; therefore, unlike Prolog, the order of the rules does not matter.

A Datalog program consists of an intensional and an extensional database. The intensional database contains the rules that are used to deduct facts (and answer queries) based on the facts in the extensional database. A Datalog rule consists of a head and a body:

```
Head :- Body.
```

In Datalog, the rule body implies the rule head.[152] The head is an atom, and the rule body is a conjunction of atoms. An atom consists of an *n-ary* predicate symbol and *n* terms (either a variable or a constant). The rule below consists of a head, an atom *king(x,y)* with two variable terms:

```
king(x,y) :- country(y), monarch(x,y), male(x) .
```

The body consists of three atoms that are in conjunction. Informally, the rule can be written as follows: "If *y* is a country, *x* is a male, and *x* is monarch of *y*, then *x* is the king of *y*." A ground fact in Datalog is a grounded rule (no variables) with an empty body. New facts can be deduced based on the existing facts and rules. Below are examples of facts:

```
country(UK) .
monarch(CharlesIII, UK) .
male(CharlesIII) .
```

Based on the facts and rule above, it can be inferred that Charles III is the king of the UK.

[152] Logically, every Datalog rule is a Horn clause.

13.3.2.3 F-Logic

Another formalism motivated by deductive databases is F-logic (Kifer 2005; Kifer and Lausen 1989).[153] F-logic was developed as an extension of Datalog with object-oriented databases in mind. Therefore, it contains primitives like classes, attributes, methods, and built-in primitives for inheritance and instantiation. More complex definitions of a domain can be done via rules.

We will explain the modelling primitives of F-logic with the help of the examples shown in Figs. 13.92, 13.93, 13.94, 13.95, and 13.96.[154] Figure 13.92 shows the subclass and instantiation relationships. In this example, *empl* is a subclass of *person*, and *faculty* is a subclass of *empl*. *umut* and *dieter* are instances of faculty, which would imply that they are also instances of *empl* and *person*.

Figure 13.93 shows the description of instances of *umut* and *dieter*. Statement (1) says that *umut* has attributes *name*, *age*, and *affiliation* defined by some classes *umut* is an element of. The *affiliation* of *umut* is also a complex object that is identified with the object id cs_1 and described with attributes *dname*, *mngr*, and *assistants*. Similarly, *dieter* in statement (2) is described with various attributes including the attribute *friends*, which is a set-valued attribute, unlike the other single-valued attributes.

Figure 13.94 shows the definition of four classes: *faculty*, *person*, *empl*, and *dept*. A class definition consists of a class name, e.g., *person*, and its signature. A signature defines the attributes and methods a class supports. For example, statement (4) says that for every instance of the class *person*, the values of *name* must be string values and the values of *friends* must be a set of person instances. An interesting feature here is depicted with the *children* property, where the range of the property is defined via a function. *child* is a unary function that takes a person instance as a parameter and defines the class that represents the set of instances of the children of a given person. Note that the range definitions in F-logic are conjunctive, that is, in statement (3), that the *boss* of a *faculty* must be an instance of the classes *faculty* and *manager* at the same time.

F-logic syntactically supports *higher-order features* that allow the usage of class and attribute names as objects.[155] Hereby, *statements over schema elements* are possible. This metamodelling approach enables useful features like the definition of values for the attributes on classes that may be inherited by the instances and subclasses of that class as default values. For example, the *highestDegree* attribute in statement (3) must have values of the type *degree*, and by default, every instance of subclasses of *faculty* (including the instances of class *faculty*) has the value *phd* for this attribute, unless it is overridden. For example, among the instance

[153] The content of this section is mainly based on Kifer et al. (1995).

[154] Adapted from Kifer et al. (1995)

[155] Note that although F-logic is syntactically higher-order, semantically, it is still strictly in first-order logic although attributes and classes can be used as objects. See Chap. 14.

$$
\begin{array}{l}
empl::person \\
faculty::empl \\
umut:faculty \\
dieter:faculty
\end{array}
$$

Fig. 13.92 Subclass and type instantiation relationships

$$
\begin{array}{l}
(1)\, umut[name \rightarrow \text{“}Umut\text{”}; \\
\qquad age \rightarrow 31; \\
\qquad affiliation \rightarrow cs_1[dname \rightarrow \text{“}CS\text{”}; \\
\qquad\qquad\qquad mngr \rightarrow dieter; \\
\qquad\qquad\qquad assistants \twoheadrightarrow \{dennis, juliette\}\,]] \\
\\
(2)\, dieter[name \rightarrow \text{“}Dieter\text{”}; \\
\qquad highestDegree \rightarrow phd; \\
\qquad friends \twoheadrightarrow \{juergen, andreas\} \\
\qquad\quad affiliation \rightarrow cs_2[dname \rightarrow \text{“}CS\text{”}\,]]
\end{array}
$$

Fig. 13.93 Database facts

$$
\begin{array}{l}
(3)\, faculty[boss \Rightarrow (faculty, manager); \\
\qquad age \Rightarrow integer; \\
\qquad highestDegree \Rightarrow degree; \\
\qquad papers \Longrightarrow article; \\
\qquad highestDegree \dashrightarrow phd; \\
\qquad avgSalary \rightarrow 50000] \\
(4)\, person\,[name \Rightarrow string; \\
\qquad friends \Longrightarrow person; \\
\qquad children \Longrightarrow child(person)] \\
\\
(5)\, empl[affiliation \Rightarrow dept; \\
\qquad boss \Rightarrow empl; \\
\qquad jointWorks@empl \Longrightarrow report] \\
\\
(6)\, dept[dname \Rightarrow string; \\
\qquad assistants \Longrightarrow (student, empl); \\
\qquad mngr \Rightarrow empl]
\end{array}
$$

Fig. 13.94 Class definitions

descriptions in Fig. 13.93, *umut* in statement (1) inherits the value *phd* for *highestDegree* attribute defined in Fig. 13.94 for the class *faculty*, while another instance may override it with the value *msc*. Classes may also contain

$$(7)\ E[boss \rightarrow M] \leftarrow E{:}empl \wedge D{:}dept \wedge E[affiliation \rightarrow D[mngr \rightarrow M{:}empl]]$$
$$(8)\ X[jointWorks@Y \twoheadrightarrow Z] \leftarrow Y{:}faculty \wedge X{:}faculty \wedge Y[papers \twoheadrightarrow Z] \wedge X[papers \twoheadrightarrow Z]$$

Fig. 13.95 Deductive rules

non-inheritable properties such as the *avgSalary* property, which makes sense, as "average salary" belongs to the class and not the instances.[156]

Like object-oriented programming languages, classes in F-logic may have methods. A method takes parameters of given types and returns a value of a given type. In fact, attributes are also methods with zero arguments. In statement (5), *jointWorks* is a method defined on the *empl* class, which takes an *empl* instance as a value and returns a set of *report* instances.

F-logic supports deductive rules to enable the definition of complex attributes and methods. Like Datalog, rules contain a body and a head, and the body implies the head of the rule. Figure 13.95 shows two deductive rules. The first one (7) is the inference rule for the values of the *boss* attribute on class *empl*. The rule can be written informally as follows: *M* is the *boss* of *E* if *M* is the *manager* of the *department* to which *E* is affiliated.

The rule in statement (8) is essentially the implementation of the method *jointWorks*. It says "give an instance of employee as a parameter; a set of papers are joint works of two faculty members if both have written those papers."

Finally, the statements in Fig. 13.96 show several queries that can be run on the database described across Figs. 13.92 to 13.95. The query in statement (9) asks for all employees in the department with the name *CS*, including their *age*, *affiliation*, and *boss*. The query would return the only instance of $umut[boss \rightarrow dieter; age \rightarrow 31; affiliation \rightarrow cs_1]$. The instance *dieter* is out because its *age* value is not known.

Similarly, statements (10) and (11) query the joint works of *dieter* from two different perspectives. The first one queries authors who have written the paper identified with *jacm90* with *dieter*; the second queries the joint works of *dieter* and *ioan*.

In a nutshell, F-logic brings the worlds of rules and object-oriented modelling together. F-logic introduces modelling primitives like classes, attributes, and different relationship types (inheritance and instantiation) to structure knowledge more ontologically. This way, F-logic merges the epistemological level and logical level as these epistemological modelling primitives have direct semantics and do not have to be mapped to a logical formalism such as first-order logic first.

There are generalizations of F-logic, such as HiLog (Chen et al. 1993), that have similar features as F-logic, like allowing metamodelling via having higher-order syntax while staying in first-order logic.

[156] Analogous to the static properties in the object-oriented programming languages.

$$
\begin{array}{l}
(9)\ ?- X{:}\,empl \wedge X[boss \rightarrow Y; \\
\qquad\qquad\qquad\quad age \rightarrow Z; \\
\qquad\qquad\qquad\quad affiliation \rightarrow D[dname \rightarrow \text{“}CS\text{”}]] \\
(10)\ ?- dieter[jointWorks@Y \twoheadrightarrow jacm90] \\
(11)\ ?- dieter[jointWorks@ioan \twoheadrightarrow Z]
\end{array}
$$

Fig. 13.96 Queries

13.3.2.4 Datalog in RIF

RIF-Core has semantically the same expressivity as Datalog. The example rule about kings in Sect. 13.3.2.2 is shown in RIF syntax in Fig. 13.97. As seen in the figure, RIF syntax is more Web-friendly, where predicates are identified with URIs and they can be grouped under namespaces.

13.3.2.5 Tools and Applications

We provide a brief overview of tools that work with rules and their application areas. We categorize tools for rules under editors, software libraries, and reasoners/rule engines. We will give some examples of tools in each category.

- In principle, any text editor can be used as an *editor* for rules as they are typically text-based. Given the nature of RIF, which is complementary to the existing Semantic Web technology, many tools for developing ontologies also have extensions for editing rules in different formats, including RIF. For example, the functionality of Protégé can be extended via plugins to support rules. An example of such a plugin is R2RIF (Pomarolli et al. 2012). R2RIF is a plugin for creating rules in RIF syntax as well as running them over RDF data. Many integrated development environments[157] also have plugins to support the syntax of Datalog.
- A *software development kit* is usually needed to work with ontologies and rules in a programming environment. A very popular one is Jena,[158] which has been maintained for many years. Alongside its widespread support for RDF(S) and OWL, it also has rule support with custom syntax.
- Finally, *reasoner* and *rule engines* allow making use of the rules in the form of inferences. For rules, there are many Datalog rule engine implementations such AbcDatalog[159] and F-logic reasoners such as FLORID (Angele et al. 2009) and Flora (Ludäscher et al. 1999). In the production rule front, there are significantly

[157] Example for VSCode: https://github.com/arthwang/vsc-prolog
[158] https://jena.apache.org/documentation/ontology/
[159] https://abcdatalog.seas.harvard.edu/

```
Document(
 Prefix(ppl <http://example.com/people#>)
 Prefix(cty <http://example.com/countries#>)
 Group
 (
  Forall ?x ?y (
     cpt:king(?x ?y) :- And(cty:country(?y) cty:monarch(?x, ?y)
      ppl:male(?x) )
  )

 )
)
```

Fig. 13.97 RIF syntax implementation of the Datalog rule about kings

more industry-grade rule engine implementations, such as Drools[160] and Jess,[161] which also have SDKs to give programmatic access to production rules.

Perhaps the most prominent applications of rules are expert systems. We already covered in Chap. 2 some of them that were implemented with production rules and logic programming languages like Prolog, which is a superset of Datalog.

An interesting application of Datalog actually comes from a field outside of deductive databases, namely, software security and quality assurance. A tool called Semmle[162] uses a version of Datalog to enable running queries over a source code to identify vulnerabilities.

As for the rule applications on the Web, RIF was implemented as an interchange format by different reasoners; for example, EYE[163] (Verborgh and de Roo 2015) is a rule-based reasoner for Linked Data that implemented RIF as an interchange format for different rule formalisms.[164]

The most prominent application of F-logic is its usage as an ontology and rule language for the Semantic Web (Kifer 2005). The implementation of OntoBroker (Fensel et al. 1998) is an application for reasoning and querying over the Semantic Web that uses F-logic as a knowledge representation formalism.

13.3.3 Summary

The ability of reasoning over data is one of the most important selling points of knowledge graphs. Although RDF and RDFS provide some mechanisms that can

[160] https://drools.org/

[161] http://alvarestech.com/temp/fuzzyjess/Jess60/Jess70b7/docs/index.html

[162] https://en.wikipedia.org/wiki/Semmle#

[163] https://josd.github.io/eye/

[164] See the rather outdated list for RIF implementations: https://www.w3.org/2005/rules/wiki/Implementations

enable reasoning, they are limited to class and property definitions and their hierarchies. For more complex reasoning, more expressive languages were needed, which led to the development of OWL and later its extension to OWL2.

In this section, we covered the OWL family of languages from an epistemological perspective. OWL contains many modelling primitives to make complex class and property definitions, such as cardinality restrictions on properties and specification of inverse properties and many more. OWL2 extends OWL with more primitives, such as qualified cardinality restrictions, property chains, and more.

OWL comes in different dialects that correspond to different subsets of first-order logic. The most popular dialect is OWL DL based on description logic, which is widely used across different applications. OWL2 takes it even further to offer different profiles of OWL DL that are optimized for different applications and purposes. As the standard ontology language for the Web, OWL has widespread tool support and many applications in different domains such as health and other life sciences, smart buildings, materials, and manufacturing.

Rules have been traditionally a popular way of representing knowledge. A clear manifestation of this is the development of expert systems. This history led to the development of a variety of rule languages with different paradigms and semantics. To bring rules on the Web, W3C published a recommendation called Rule Interchange Format (RIF) as an intermediate format that enables semantics preserving mappings from production and logic rules from different languages to each other. The RIF format reuses the technologies from the Semantic Web Stack and is compatible with RDF and OWL, which makes it suitable as a format to publish rules on the Web. RIF contains many dialects that correspond to different intersecting sets of rule formalisms. Three of these dialects are rather well specified, namely, RIF-PRD, RIF-BLD, and their intersection RIF-Core. RIF takes its roots syntactically and semantically from rule languages for deductive databases such as Datalog and F-logic. Datalog is a logic programming language developed for deductive databases for relational model, and F-logic is a language for deductive object-oriented databases with metamodelling features.

From a formal perspective, languages like OWL and rule languages represent two camps for knowledge representation: OWL is mainly based on description logic which has standard first-order logic semantics with OWA and not UNA. Datalog and F-logic are more oriented toward minimal model semantics and consequently the CWA and UNA. In Chap. 14, we will focus on these logical formalisms in more detail.

13.4 SKOS: A Lightweight Approach for Schema Integration

SKOS has been designed to provide a low-cost migration path for porting existing organization systems to the Semantic Web. SKOS also provides a lightweight, intuitive conceptual modeling language for developing and sharing new KOSs. It can be used on its own, or in combination with more-formal languages such as the Web Ontology Language (OWL)

> SKOS can also be seen as a bridging technology, providing the missing link between the rigorous logical formalism of ontology languages such as OWL and the chaotic, informal and weakly-structured world of Web-based collaboration tools, as exemplified by social tagging applications.—W3C SKOS Primer[165]

The *SKOS (Simple Knowledge Organization System)* is a common data model for representing, sharing, and linking knowledge organization systems (KOS) via the Semantic Web. It has mainly two purposes:

- Providing a lightweight mechanism for aligning ontologies and building ontology networks
- Bridging the gap between formal ontologies and the informal nature of the Web

In this section, we first introduce various types of knowledge organization systems and then explain different modelling primitives of SKOS that provide a unified way of representing different KOS. Finally, we briefly introduce tools for applications of SKOS and provide a summary at the end.[166]

13.4.1 Knowledge Organization Systems

There are different types of knowledge organization systems with various levels of expressivity (Fig. 13.98). Before we start explaining SKOS, we will first briefly introduce these systems that are the foundation of the SKOS vocabulary.

13.4.1.1 Controlled Vocabulary

Controlled vocabularies[167] are the most generic knowledge organization systems. They are typically just a collection of canonical terms used for different concepts without any indication of how they are organized. The vocabulary is controlled in the sense that the terms used are determined by a group of people in consensus. It aims at eliminating the ambiguities caused by, e.g., spelling variants of the terms describing the same concept. The higher we go on the expressivity ladder (toward thesauri), the more organized a controlled vocabulary becomes. Every knowledge organization system is essentially based on a controlled vocabulary.

[165] https://www.w3.org/TR/skos-primer/

[166] See also Bechhofer (2010) and Isaac (2011).

[167] https://en.wikipedia.org/wiki/Controlled_vocabulary

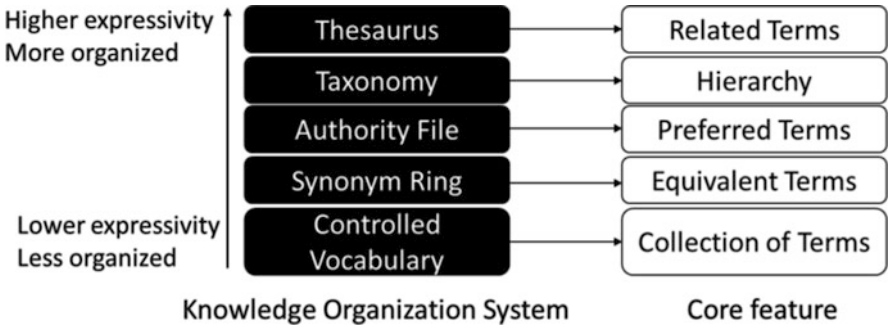


Fig. 13.98 Knowledge organization systems and their classification [Figure adapted from Bechhofer (2010)]

13.4.1.2 Synonym Rings

Synonym rings[168] are a knowledge organization system that specifies the semantic equivalence relationship between different terms. A synonym ring is also known as a *synset*. For example, WordNet (Fellbaum 2010) is mainly organized around synsets. Figure 13.99 shows how "city" is defined in WordNet.[169] The synonyms of city are listed next to it (metropolis, urban center).

13.4.1.3 Authority Files

An authority file is a controlled vocabulary that indicates a preference between different names referring to the same concepts. Figure 13.100 shows an authority file from the German National Library about Qaddafi. It lists the preferred name (e.g., Qaddafi, Muammar al-) for the concept and the other names (under "Andere Namen").

13.4.1.4 Taxonomies

Taxonomies organize concepts in a hierarchical manner. They provide a subsumption relationship between concepts. Taxonomies are heavily used in biology and other life sciences, library science, Web search, e-commerce, and many other fields. Figure 13.101 shows the classification of animals according to Aristotle.

[168] https://en.wikipedia.org/wiki/Synset

[169] http://wordnetweb.princeton.edu/perl/webwn?s=city

Synset: (noun) city, metropolis, urban center (a large and densely populated urban area; may include several independent administrative districts) *"Ancient Troy was a great city."*

Fig. 13.99 The synonym ring for "city" in WordNet

GND	
Link zu diesem Datensatz	https://d-nb.info/gnd/118559060
Person	Qaḏḏāfī, Muʿammar al-
Geschlecht	männlich
Andere Namen	Kaddafi Kadhafi, Muammar al- Kadhafi, Muammmar al- Qadāfī, Muʿammar al- Gaddafi, Umar Muammar al- Gaddafi, Muammar al- Qaḏḏāfī, ʿUmar Muʿammar al- Kadhdhafi, Umar Muammar al- Kadhdhafi, Muammar al- Gaddhafi, Moamar al- Gaddhafi, Muʿammar al- Al-Qadhafi, Muammar Qadhafi, Muammar al- Qaddhafi, Muammar el- Al-Kadhafi, Muammar Kaddafi, Muammar

Fig. 13.100 Authority file from the German National Library for Qaddafi (https://d-nb.info/gnd/118559060)

13.4.1.5 Thesaurus

A thesaurus extends a taxonomy with various types of relationships. Alongside hierarchy relationships, thesauri also support relationships like synonymity, narrowness, broadness, and relatedness in generic sense. Figure 13.102 shows the concept of "aircraft spin" from the NASA Thesaurus.[170] It provides a definition of the term and other related terms.

[170] https://sti.nasa.gov/docs/thesaurus/thesaurus-vol-1.pdf

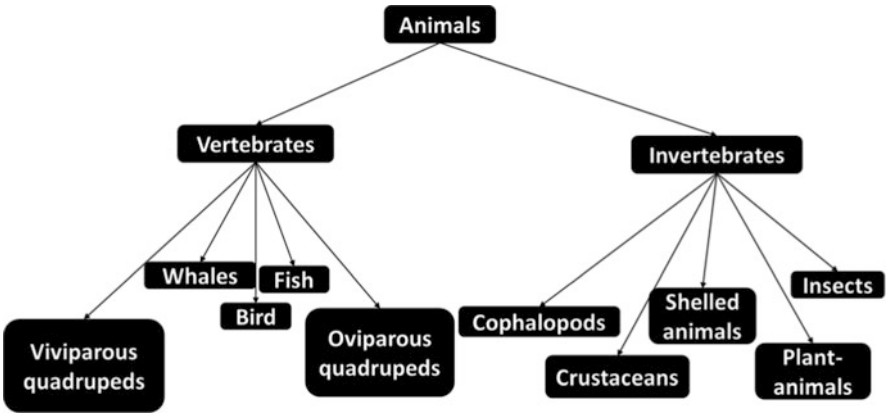


Fig. 13.101 Aristotle's classification of animals (Adapted from https://sites.google.com/site/aristotlethebiologist/aristotle-s-biology/classification)

13.4.2 Simple Knowledge Organization System (SKOS)

The Simple Knowledge Organization System (SKOS) is a data model for sharing knowledge organization systems on the Web. It is a W3C recommendation[171] that aims to bridge the gap between existing knowledge organization systems and the Semantic Web. There is a plethora of standards for organizing knowledge from different fields, e.g., library and life sciences. SKOS allows interoperability between these different standards and conventions. The main goals of SKOS are (Bechhofer 2010):

- To provide a simple, machine-understandable model for knowledge organization systems (KOS)
- That is flexible and extensible enough to deal with the heterogeneity of KOS
- That can support the publication and use of KOS within a decentralized, distributed information environment such as the World Wide (Semantic) Web

In the following, we will introduce the main modelling primitives of SKOS for representing concepts from different knowledge organization systems. Since SKOS uses the RDF model, the examples will be given in Turtle RDF serialization format.

[171] This section contains content derived/copied from the latest version of the SKOS Reference https://www.w3.org/TR/skos-reference (last accessed September 2022). The examples are based on the SKOS Primer https://www.w3.org/TR/skos-primer/ and SKOS Reference document https://www.w3.org/TR/skos-reference. Copyright © 2009 World Wide Web Consortium (MIT, ERCIM, Keio, Beihang). http://www.w3.org/Consortium/Legal/2015/doc-license

aircraft spin
DEF A prolonged stall in fixed-wing aircraft characterized by a sustained spiral descent, usually with the nose down.
RT aerodynamic stalling
∞ aircraft
aircraft performance
aircraft safety
control stability
controllability
crash landing
flight hazards
flight safety
hazards
maneuvers
spin dynamics

Fig. 13.102 Aircraft spin concept from the NASA Thesaurus

13.4.2.1 Concept

A SKOS concept can be viewed as an idea or notion: a unit of thought. However, what exactly comprises a unit of thought is not formally defined, and this definition is meant to be suggestive rather than restrictive. A SKOS concept is an instance of *skos:Concept*. The example in Fig. 13.103 defines the *:Cat* concept.

13.4.2.2 Concept Scheme

A SKOS concept scheme can be seen as an aggregation of one or more SKOS concepts. Semantic relationships (links) between those concepts may also be seen as part of a concept scheme. A concept scheme is an instance of *skos:ConceptScheme*. There are three properties related to concept schemes:

- *skos:inScheme*: connects a resource with a concept scheme.
- *skos:hasTopConcept* and *skos:topConceptOf*: specify the topmost concept in a concept scheme. There is, however, no formal integrity constraint that forbids a top concept to have a broader concept.

Figure 13.104 shows an example of a concept scheme about living things. It defines that *ex:LivingThing* is the topmost concept in the scheme, where the concept *ex:Mammal* is a part of the scheme.

```
ex:Cat rdf:type skos:Concept .
```

Fig. 13.103 A SKOS concept

```
ex:LivingThings rdf:type skos:ConceptScheme ;
        skos:hasTopConcept ex:LivingThing .
        ex:LivingThing skos:topConceptOf ex:LivingThings .
        ex:Mammal skos:inScheme ex:LivingThings .
```

Fig. 13.104 A SKOS concept scheme

13.4.2.3 Lexical Label

A lexical label is a string of UNICODE characters, in an optionally specified natural language. SKOS provides three properties for specifying lexical labels:

- *skos:prefLabel*: the preferred label of a resource
- *skos:altLabel*: the alternative label of a resource
- *skos:hiddenLabel*: the label that is invisible to the user but may be used in the application for some purpose (e.g., to represent a misspelled variant)

The three lexical label properties are pairwise disjoint, and one concept cannot have more than one preferred label with the same language tag. Figure 13.105 shows various lexical labels for a concept called Animals.

13.4.2.4 Documentation Properties

Documentation properties are used to provide information about SKOS concepts. There is no restriction on the format and purpose of this information, e.g., it could be plain text, hypertext, or an image; it could be a definition, information about the scope of a concept, editorial information, and so on.

There are seven documentation properties provided by SKOS; however, there is no strict guideline on how they can be used. The most generic one is *skos:note*, which has six subproperties:

- skos:changeNote
- skos:definition
- skos:editorialNote
- skos:example
- skos:historyNote
- skos:scopeNote

Figure 13.106 shows an example of documentation of the ex:*Mammal* concept.

```
ex:Animals
        skos:prefLabel "animals"@en ;
        skos:altLabel "fauna"@en ;
        skos:hiddenLabel "aminals"@en ;
        skos:prefLabel "animaux"@fr ;
        skos:altLabel "faune"@fr .
```

Fig. 13.105 SKOS lexical labels

```
ex:Mammal skos:note “a note about the concept"@en .
```

Fig. 13.106 A SKOS note

13.4.2.5 Semantic Relations

SKOS semantic relations are links between SKOS concepts. SKOS has two basic categories of semantic relations, hierarchical relations and associative relations.

Hierarchical relations organize concepts in a hierarchy. There are two properties for defining such relations:

- *skos:broader*: A direct hierarchical link between two concepts. *A skos:broader B* means that *B* is a broader concept than *A*.
- *skos:narrower*: A direct hierarchical link between two concepts. *C skos:narrower D* means that *D* is a narrower concept than *C*.

Figure 13.107 shows an example with two concepts, *ex:mammals* and *ex:animals*, where *ex:animals* is narrower than *ex:mammals* and *ex:mammals* is defined as broader than *ex:animals*.

Note that *skos:broader* and *skos:narrower* properties are not transitive and are only used to represent direct links between concepts. SKOS also offers a transitive version of these properties, namely, *skos:broaderTransitive* and *skos:narrowerTransitive*.

Associative relations organize concepts without a particular hierarchy or a specific nature of the relatedness. *skos:related* property is used to define a direct associative link between two concepts. It indicates that the concepts are somewhat related but makes no assumptions about the nature of the relatedness. *skos:related* is not transitive. Figure 13.108 is an example of two concepts that are somehow related to each other.

There are some important characteristics of hierarchical and associative relations.

- *skos:related* is a symmetric property.
- *skos:broaderTransitive* and *skos:narrowerTransitive* are each transitive properties.
- *skos:narrower* is an inverse property of *skos:broader*.
- *skos:narrowerTransitive* is an inverse property of *skos:broaderTransitive*.

```
ex:animals rdf:type skos:Concept;
        skos:prefLabel "animals"@en;
        skos:narrower ex:mammals.
ex:mammals rdf:type skos:Concept;
        skos:prefLabel "mammals"@en;
        skos:broader ex:animals.
```

Fig. 13.107 SKOS hierarchical relations

```
ex:birds rdf:type skos:Concept;
        skos:prefLabel "birds"@en;
        skos:related ex:ornithology.
        ex:ornithology rdf:type skos:Concept;
        skos:prefLabel "ornithology"@en.
```

Fig. 13.108 SKOS associative relation

- Associative and hierarchical relations are strictly disjoint: *A skos:broader B* . *A skos:related B* would be an illegal definition. The same goes for the transitive versions of the hierarchical relations.

There are many other characteristics regarding semantic relations (e.g., allowing cycles in semantic relations), which can be found in the SKOS specification.

13.4.2.6 Mapping Properties

The mapping properties are used to link SKOS concepts in different concept schemes. These are the following properties offered by SKOS as mapping properties:

- *skos:mappingRelation*: the most generic mapping property. Super property of all other mapping properties.
- *skos:broadMatch* and skos:*narrowMatch* specify a hierarchical relationship between two concepts, in a similar fashion as *skos:broader* and *skos:narrower*, respectively.
- *skos:relatedMatch* specifies an associative relationship akin to *skos:related*.
- skos:closeMatch specifies a link between two concepts that are "similar enough" and indicates that these concepts *may* be used interchangeably**.** The property is not transitive to avoid potential propagation of mapping errors across multiple concept schemes.
- *skos:exactMatch* links two concepts to indicate that these concepts *can be considered the same with a high degree of confidence.* The property is transitive and a subproperty of *skos:closeMatch.*

```
ex1:platypus skos:broadMatch ex2:eggLayingAnimals.
ex1:platypus skos:relatedMatch ex2:eggs.
ex1:animal skos:exactMatch ex2:animals.
```

Fig. 13.109 SKOS mapping properties

The example in Fig. 13.109 shows three concept mappings across different concept schemes represented with *ex1* and *ex2* namespaces.

Note that although conventionally the mapping properties are used between concepts from different concept schemes, there is no strict constraint enforced by SKOS on this. Mapping two concepts from the same concept scheme would not be contradicting with the SKOS specification.

skos:exactMatch can be seen as a lightweight alternative to *owl:sameAs*. *skos:Concept* instances are viewed as individuals in terms of OWL; therefore, they can be linked with *owl:sameAs*; however, this would imply a strong commitment to the equality of these concepts. All property value assertions on each concept must hold for both. For example, two concepts linked with *owl:sameAs* may have multiple *skos:prefLabel* values with the same language tag, which is not allowed by the SKOS specification. *skos:exactMatch* does not enforce such a strong notion of equality; see also Halpin et al. (2010).

13.4.3 Tools and Applications

In the following, we provide a short list of SKOS tools and applications. Many other tools (Isaac 2011) and applications[172] can be found online.

As the SKOS model is based on existing technologies from the Semantic Web Stack, almost all tools that can be used for RDF(S) and OWL can also be used for modelling knowledge with SKOS. For example, Protégé is already a widespread tool to work with SKOS, often even together with OWL ontologies. Since a SKOS concept scheme is encoded in RDF syntax, any RDF visualization tool and browser can be used.

There are, however, still tools dedicated to creating knowledge organization systems like thesaurus or taxonomy with the SKOS model. PoolParty[173] is a prominent one that provides a GUI application including editing, browsing, and verification features for knowledge represented with SKOS. Similarly, Skosmos[174] provides an open-source tool for publishing and browsing SKOS concept schemes.

SKOS has found widespread application in Linked Open Data. One of the motivations of SKOS was to bring existing knowledge organization systems into

[172] https://www.w3.org/2001/sw/wiki/SKOS/Datasets

[173] https://www.poolparty.biz/skos-and-skos-xl

[174] https://skosmos.org/

Linked Open Data. Many LOD datasets use SKOS to some extent to align themselves with other datasets. For example, DBpedia links the resource *dbpedia:Innsbruck* with WordNet with the help of the skos:exactMatch property.[175]

An application from e-commerce of SKOS comes from the GS1 organization. GS1 is the international organization that manages the barcode standards. They also provide the GS1 vocabulary that contains the terms used across GS1 standards. This vocabulary is an OWL ontology but also aligned with schema.org with SKOS properties.[176]

Our final example application is from the US Library of Congress**.** The Library of Congress publishes its catalog as Linked Data, and the classification subjects are organized as a SKOS vocabulary.[177]

13.4.4 Summary

The need for organizing and representing knowledge goes way before knowledge graphs, the Semantic Web, the Web, or even computers. By the time technologies like Linked Data and the Semantic Web were developed, there was already a plethora of knowledge organized on the Web rather informally with different expressivity. This led to a significant amount of heterogeneity between different models and their encodings and created a barrier between the Semantic Web and existing taxonomies, thesauri, and alike.

SKOS provides a homogeneous model for representing various knowledge organization systems on the Web and allows bridging the gap between informal ways of representing knowledge on the Web and formal ontologies. In this section, we mainly focused on the epistemological aspect of SKOS and introduced its modelling primitives and how these primitives are related to each other.

We also provided a sample list of tools and applications, which takes some examples from the very widespread tool support and adoption by applications in different domains. This is not a coincidence, as SKOS is built on top of the existing semantic technologies, which reduces the entry barrier as existing tools for the Semantic Web can be reused.

SKOS does not have direct formal semantics; however, it is somewhat aligned with RDF and OWL. Any RDF serialization syntax can be used to serialize a SKOS graph as all SKOS graphs are valid RDF graphs. Formally, SKOS is an OWL ontology, and a specific SKOS vocabulary (e.g., a specific taxonomy) is an instantiation of the SKOS ontology. *skos:Concept* is an instance of *owl:Class,* and individual concepts are instances of this class.

[175] https://dbpedia.org/page/Innsbruck

[176] Example shows the Brand type: https://www.gs1.org/voc/Brand.

[177] https://loc.gov

The SKOS data model is in OWL Full. However, there are several ways to interplay SKOS vocabularies with OWL ontologies and stay on the decidable side of OWL.[178]

The relationship between OWL and SKOS (therefore its semantics) is only loosely defined. The primary goal of SKOS is not to enable the inference of new knowledge but to represent the knowledge in an existing KOS on the Semantic Web.

13.5 Summary

The technologies that are used for building knowledge graphs take their roots from the previous work on knowledge representation and the Semantic Web. Many languages that are initially developed within the Semantic Web Stack are also used to model knowledge graphs. In this chapter, we covered them in four aspects:

- Data modelling
- Data retrieval and manipulation
- Reasoning
- Bridging formal and informal knowledge

For modelling the data for knowledge graphs, RDF and RDFS provide the main data model and building blocks. They offer minimal practical means to describe data in a knowledge graph semantically, namely, instances, types, properties, and their hierarchies.

The data modelled in RDF is queried with SPARQL. Aligned with the triple-based model of RDF, SPARQL works via matching given triple patterns with the triples in an RDF dataset. The retrieval queries can return results that are bound to the variables in the triple patterns or even entire subgraphs. It supports functions to process and aggregate the query results. For the manipulation of the data in a knowledge graph, SPARQL also provides queries for inserting and deleting data.

Another way of manipulation is done by applying constraints on an RDF dataset. This is typically challenging with the Semantic Web Stack as the technologies there mainly have the open-world assumption. Therefore, constraints on RDF graphs were traditionally defined as SPARQL queries in an ad hoc way since querying is inherently done under the closed-world assumption. To make constraint definitions more declarative to improve reusability and maintainability, SHACL was published as a W3C recommendation. With SHACL, different kinds of constraints can be defined in a declarative way (e.g., cardinality, value type, property pair constraints). The constraints can be then implemented as SPARQL queries to run over RDF graphs.

[178] See https://www.w3.org/2006/07/SWD/SKOS/skos-and-owl/master.html. Also, remember Punning in OWL2 (Sect. 13.3.1.2).

One of the most important advantages of using a knowledge graph instead of an RDB or a traditional graph database is the ability to make implicit knowledge explicit via reasoning. RDFS already provides some mechanisms for the subsumption of types and inference of instance types. For more complex systems, something beyond RDFS was needed, which led to the development of OWL, a knowledge representation language based on description logic. Besides languages like OWL, knowledge can be also represented with rules and facts, which was the most prominent way for many years in the development of AI. Over the years, many different rule systems and formalisms were developed. To bring these on the Web, RIF was developed as part of the Semantic Web Stack. RIF is a language framework to represent rules from different paradigms and expressivity.

Languages like RDFS, OWL, and various rule languages have formal underpinnings that enable reasoning by machines. However, there are also *less strictly formal* ways of organizing knowledge, and they are quite widespread. SKOS was published as a model to bridge the gap between this formal world of ontologies and the knowledge organization systems like taxonomies and thesauri.

In this chapter, we focused on the epistemological aspects of these languages. One advantage of using existing Semantic Web Stack technologies while building knowledge graphs is that they are mostly standardized by bodies like W3C. Consequently, each of the languages has a rather wide variety of tools to work with them and applications that use them. Although arguably that it would be part of the implementation level and not epistemology, we also introduced various tools and applications for each language we covered in this chapter.

In the next chapter, we will describe the main logical formalizations that form the underpinnings of the languages described here.

References

Angele J, Kifer M, Lausen G (2009) Ontologies in F-logic. In: Studer R (ed) Staab S. Springer, Handbook on ontologies, pp 45–70

Antoniou G, Groth P, Van Harmelen F, Hoekstra R (2012) A semantic web primer, 3rd edn. MIT press

Bechhofer S (2010) SKOS: past, present and future. The Semantic Web: Research and applications. http://videolectures.net/eswc2010_bechhofer_sppf/

Carroll JJ, Bizer C, Hayes P, Stickler P (2005) Named graphs. Journal of Web Semantics 3(4): 247–267

Ceri S, Gottlob G, Tanca L et al (1989) What you always wanted to know about Datalog (and never dared to ask). IEEE Trans Knowl Data Eng 1(1):146–166

Chen W, Kifer M, Warren DS (1993) HiLog: a foundation for higher-order logic programming. J Log Program 15(3):187–230

Corby O, Zucker CF, Gandon F (2014) SPARQL template: a transformation language for RDF. PhD thesis, Inria

De Giacomo G, Lenzerini M, Rosati R. On higher-order description logics. In Proceedings of the 22nd international workshop on description logics (DL2009) Oxford, July 27–30, 2009

Dumontier M (2012) Real world applications of OWL, Protege short course presentation. https://www.slideshare.net/micheldumontier/owlbased-applications

Fellbaum C (2010) Princeton university: about WordNet. https://wordnet.princeton.edu/
Fensel D, Decker S, Erdmann M, Studer R (1998) Ontobroker: the very high idea. In: Proceedings of the eleventh international Florida artificial intelligence research society conference, May 18–20, 1998, AAAI Press, Sanibel Island, FL, pp 131–135
Fensel D, Van Harmelen F, Horrocks I, McGuinness DL, Patel-Schneider PF (2001) OIL: an ontology infrastructure for the semantic web. IEEE Intell Syst 16(2):38–45
Gayo JEL, Prud'Hommeaux E, Boneva I, Kontokostas D (2017) Validating RDF data. Synthesis Lectures on Semantic Web: Theory and Technology 7(1):1–328
Halpin H, Hayes PJ, McCusker JP, McGuinness DL, Thompson HS (2010) When owl: sameAs isn't the same: an analysis of identity in linked data. In: The semantic web–ISWC 2010: 9th international semantic web conference, ISWC 2010, Shanghai, November 7–11, 2010, Revised Selected Papers, Part I, Springer, pp 305–320
Hammond T, Pasin M, Theodoridis E (2017) Data integration and disintegration: managing springer nature SciGraph with SHACL and OWL. In: Proceedings of the international semantic web conference (ISWC) 2017 (Posters, Demos & Industry Tracks), October 21–25, Vienna
Hartig O, Champin PA (2021) Metadata for RDF statements: the RDF-Star approach. In: Lotico
Herman I (2012) SW tutorial. http://www.w3.org/People/Ivan/CorePresentations/SWTutorial
Horrocks I (2009) OWL 2: the next generation. London Semantic Web Meetup Group. http://www.cs.ox.ac.uk/people/ian.horrocks/Seminars/download/OWL2-overview.pdf
Isaac A (2011) SKOS: simple knowledge organization system. Dublin Core tutorial. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.9073&rep=rep1&type=pdf/
Käfer T (2021) Distributed knowledge graphs: knowledge graphs and linked data, AI4Industry Summer School. https://ai4industry2021.sciencesconf.org/data/program/dkg.pdf
Kifer M (2005) Rules and ontologies in F-logic. In: Reasoning Web: First International Summer School 2005, Msida, Malta, July 25–29, 2005, Revised Lectures, Springer, pp 22–34
Kifer M (2011) KR and reasoning on the semantic web: RIF. In: Domingue J, Fensel D, Hendler JA (eds) . Springer, Handbook of semantic web technologies, pp 399–439
Kifer M, Lausen G (1989) F-logic: a higher-order language for reasoning about objects, inheritance and schema. In: SIGMOD/PODS04: international conference on management of data and symposium on principles database and systems, Portland, OR, June 1, pp. 134–146
Kifer M, Lausen G, Wu J (1995) Logical foundations of object-oriented and frame-based languages. J ACM 42(4):741–843
Kontokostas D, Westphal P, Auer S, Hellmann S, Lehmann J, Cornelissen R, Zaveri A (2014) Test-driven evaluation of linked data quality. In: Proceedings of the 23rd international conference on World Wide Web, Seoul, Korea, April 7–11, pp 747–758
Ludäscher B, Yang G, Kifer M (1999) Flora: The secret of object-oriented logic programming. Technical report, SUNY at Stony Brook
McGuinness D, Fikes R, Stein L, Hendler J (2003) DAML-ONT: an ontology language for the Semantic Web, pp 65–93. In: Fensel et al. [2005]
Pomarolli A, Anderlik S, Kŭng J (2012) R2RIF-rule integration plugin for Protégé OWL. In: Computer aided systems theory–EUROCAST 2011: 13th international conference, Las Palmas de Gran Canaria, Spain, February 6–11, 2011, Revised Selected Papers, Part I 13, Springer, pp 137–144
Rietveld L, Hoekstra R (2017) The YASGUI family of SPARQL clients. Semantic Web 8(3): 373–383
Sack H (2020) Semantic Web Technologies. OpenHPI, Hasso Plattner Institute. https://www.youtube.com/playlist?list=PLoOmvuyo5UAeihlKcWpzVzB51rr014TwD
Straccia U (2013) Foundations of fuzzy logic and semantic web languages. Taylor & Francis
Verborgh R, De Roo J (2015) Drawing conclusions from linked data on the web: the EYE reasoner. IEEE Softw 32(3):23–27

Chapter 14
The Logical Level



Meanwhile, we have introduced the epistemological level, which should bridge the human understanding of a knowledge graph with its computational meaning. For this, we need to define a formal meaning for the used modelling constructs as a specification of whether and how an implementation should understand and manipulate a knowledge graph. We do this in three steps.

- *Data reasoning*: we introduce mathematical logic and variations like descriptions logic and Herbrand model semantics.
- *Data modelling*: we discuss the formal semantics for the data model we are using. RDF(S) adds complexity by using a graph model. We will see that this makes it difficult to combine it with standard logical languages.
- *Data retrieval and update*: we discuss the semantics of data retrieval and manipulation. We have to rely on a different mathematical segment, i.e., algebras, and we will show how it can be used to define the formal semantics of SPARQL operators.

Finally, we are providing a summary of the chapter. In the context of this book, we only cover a core piece of each logical formalism. We refer the readers to the books like Ben-Ari (2012) and Schöning (2008) and the *Introduction to Logic* lecture of Stanford[1] for more detailed coverage of each topic, from which we also benefited for the preparation of the chapter.

[1] http://intrologic.stanford.edu/

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_14

14.1 Logics

Logic is the study of entailment relations. It deals with the question of how conclusions follow from premises defined with formal language. Knowledge graphs must be represented at the logical level to enable their machine understandability. Therefore, it is natural to take tools and techniques from logic while building and using knowledge graphs: e.g., inferring new statements, given a set of explicit statements in a knowledge graph.

In this section, we will introduce various logical languages, mainly their syntax, interpretation, model theory, and proof systems. It is important to understand these fundamental formalisms as they provide the formal means for machines to understand many of the modelling languages we introduced in Chap. 13. We will first start with propositional logic and then cover first-order logic, which addresses the limitations of propositional logic in the expense of decidability. We then cover two modifications of first-order logic with more favorable computational properties: *description logic (DL)*, which defines subsets of first-order logic, followed by restricting and allowing only certain interpretations (*Herbrand interpretations*) and specific models (minimal models). We then briefly cover second-order logic and other variants of logic including adaption to Web-scale reasoning and finally provide a summary.

14.1.1 Propositional Logic

In this section, we introduce propositional logic, particularly its motivation, its syntax, and how to reason with propositional logic (interpretations, model theory, proof systems[2]). We introduce an inference rule called resolution with an example, and finally, we provide a summary.

14.1.1.1 Motivation

Propositional logic can be used to assign truth values to statements (propositions). For the sake of conciseness, propositions are represented with symbols, and they can be combined with different connectors. For example:

- p represents "It is raining."
- q represents "The sun is shining."
- r represents "There will be a rainbow."
- $(p \wedge q) \rightarrow r$ represents "If it is raining and the sun is shining, then there will be a rainbow."

[2] See also http://intrologic.stanford.edu/chapters/chapter_04.html.

Propositional logic is one of the simplest and most common logic and lies at the core of all other logical formalisms.

14.1.1.2 Syntax

The syntax of propositional logic consists of the following elements:

- Logical constants: *true*, *false*.
- Propositional symbols: p, q, s, etc.
- Parentheses wrapping around formulas: (. . .).
- Atomic formulas, which are propositional symbols or logical constants.
- Formulas are either atomic formulas or can be formed by combining atomic formulas with the following connectives:
  - Conjunction connector (and) $\wedge$
  - Disjunction connector (or) $\vee$
  - Implication connector (implies) $\rightarrow$
  - Biconditional connector (is equivalent) $\leftrightarrow$
  - Negation operator (not) $\neg$

Ambiguities for the order of evaluation in a formula are resolved through the precedence of the connectives. The connectors are evaluated in the following order: negation, conjunction, disjunction, implication, and bidirectional. Additionally, precedence can be defined via parentheses: e.g., the formula $\neg p \vee q \wedge r$ is equivalent to $(\neg p) \vee (q \wedge r)$.

Given these syntactic elements, a well-formed formula in propositional logic is defined based on the following syntax rules:

- A logical constant or a propositional symbol is a well-formed formula.
- If f is a well-formed formula, then $\neg f$ is a well-formed formula.
- If f is a well-formed formula, then (f) is a well-formed formula.
- If f_1 and f_2 are well-formed formulas, then $(f_1 \wedge f_2)$, $(f_1 \vee f_2)$, $(f_1 \rightarrow f_2)$, and $(f_1 \leftrightarrow f_2)$ are well-formed formulas.
- A well-formed formula results from a finite number of applications of the above rules.

For example, the following are well-formed formulas: $\neg p \wedge \neg q$, $\neg(p \wedge q)$, and $(((\neg p) \wedge q) \rightarrow r) \leftrightarrow ((\neg r \wedge p))$.

14.1.1.3 Interpretation

An interpretation in logic deals with the assignment of truth values to symbols. In propositional logic, an interpretation is a function I that maps all propositions of a logical theory (set of formulas) to truth values F or T with $I(\text{false}) = F$ and $I(true) = T$. The truth value of complex formulas is assigned recursively:

$$
\begin{aligned}
&I(f) = I((p \rightarrow q) \leftrightarrow (p\ V\ q))\\
&= I(p \rightarrow q) \leftrightarrow I(p\ V\ q)\\
&= (I(p) \rightarrow I(q)) \leftrightarrow (I(p)\ V\ I(q))\\
&= (I(p) \rightarrow I(q)) \leftrightarrow (I(p)\ V\ I(q))\\
&= (F \rightarrow T) \leftrightarrow (F\ V\ T)\\
&= (F \rightarrow T) \leftrightarrow T\\
&= T \leftrightarrow T\\
&= T
\end{aligned}
$$

Fig. 14.1 Calculation of the truth value for the formula f

- $I(f_1 \wedge f_2) = T$ if and only if $I(f_1) = I(f_2) = T$; F otherwise.
- $I(f_1 \vee f_2) = T$ if and only if $I(f_1) = T$ or $I(f_2) = T$; F otherwise.
- $I(f_1 \rightarrow f_2) = T$ if and only if $I(f_1) = F$ or $I(f_2) = T$; F otherwise.
- $I(f_1 \leftrightarrow f_2) = T$ if and only if $I(f_1) = I(f_2)$; F otherwise.

The truth value of a logical theory is T if and only if all its elements are evaluated to T; F otherwise.

Let us demonstrate how interpretations assign a truth value to a formula, given the interpretations of individual proposition symbols. Suppose I is an assignment for which $I(p) = F$ and $I(q) = T$. If $f = (p \rightarrow q) \leftrightarrow (p\ \mathrm{V}\ q)$, what is $I(f)$?

Given the interpretations of p and q, as well as the precedence rules of the connectors, *I(f)* can be calculated, as shown in Fig. 14.1.

14.1.1.4 Model Theory

Model theory deals with the interpretations that make a set of formulas true. A propositional formula f is *satisfiable* if and only if $\mathrm{I}(f) = T$ in some interpretation I. Such an interpretation is called a *model* of f. S is *unsatisfiable* (or contradictory) if it is false in every interpretation. A set of finite formulas $S = \{f_1, f_2, \ldots, f_n\}$ is satisfiable if and only if there exists an interpretation I such that $I(f_1) = I(f_2) = \ldots = I(f_n) = T$; such an interpretation is called a model of S. S is unsatisfiable if no such interpretation exists.

There are certain properties of satisfiability that may make life easier while calculating whether an interpretation is a model of a set of formulas (Ben-Ari 2012):

- If S is satisfiable, then so is $S - \{f_i\}$ for any $i = 1, 2, \ldots, n$.
- If S is satisfiable and f is a *valid formula*, i.e., if every interpretation of it is a model of it, then $S \cup \{f\}$ is also satisfiable.
- If S is unsatisfiable and f is any formula, $S \cup \{f\}$ is also unsatisfiable.

The most important application of model theory is the logical consequence relationship. This relationship is denoted by the double turnstile symbol. $f_1 \vDash f_2$ denotes that f_2 is a logical consequence of f_1, if and only if every model of the set of formulas f_1 is also a model of the set of formulas f_2. If the inverse of this relationship

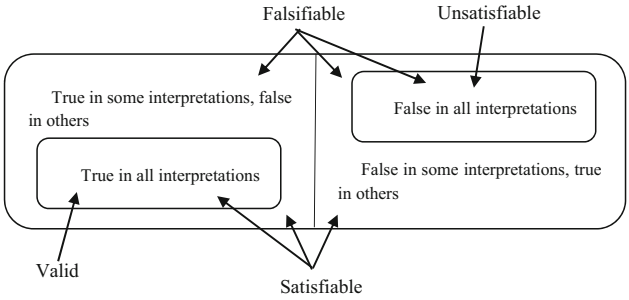


Fig. 14.2 The relationship between valid, satisfiable, falsifiable, and unsatisfiable formula (Adapted from Fig. 2.6 in Ben-Ari (2012))

also holds ($f_2 \vDash f_1$), i.e., every model of f_2 is also a model of f_1, then these two sets of formulas are logically equivalent ($f_1 \equiv f_2$).

In model theory, in addition to satisfiable and unsatisfiable formulas, we can also talk about valid and falsifiable formulas (Ben-Ari 2012). Given a set of formulas S, S is *valid* (a tautology), denoted $\vDash S$, if and only if $I(S) = T$, for all interpretations I (e.g., "It's raining or it's not raining"). S is not valid (*falsifiable*), if we can find some interpretation I, such that $I(S) = F$. Relationship between *validity*, *satisfiability*, *falsifiability*, and *unsatisfiability* is depicted in Fig. 14.2.

We can check whether a set of formulas S_2 is a logical consequence of a set of premises S_1 by building the truth table for the logical constants of the language. The truth table method can be used to determine any property of a logical theory in propositional logic that is related to the logical consequence relationship (e.g., if it is a tautology).

The truth table method can be formalized as follows:

1. Build the truth table for the premises.
2. Build a truth table for the conclusion.
3. Finally, compare the two tables to see if every interpretation (truth assignments to the propositional symbols) in the truth table from the first step that makes the premises true also makes the conclusion true.

Let us demonstrate the evaluation of logical consequence with the truth table method. Consider the following simple propositions.

- It is warm outside: p.
- The weather is nice: q.
- John goes to the beach: r.

Now, consider the following set of formulas S_1 built with the proposition above as premises:

- If it is warm outside, then the weather is nice: $p \rightarrow q$.
- If John goes to the beach, it is warm outside, or the weather is nice: $r \rightarrow p \vee q$.

Table 14.1 The truth table for the premises

p	q	r	$(p \rightarrow q)$	$(r \rightarrow p \vee q)$	$\boldsymbol{S_1}$
T	T	T	T	T	**T**
T	T	F	T	T	**T**
T	F	T	F	T	**F**
T	F	F	F	T	**F**
F	T	T	T	T	**T**
F	T	F	T	T	**T**
F	F	T	T	F	**F**
F	F	F	T	T	**T**

Table 14.2 The truth table in Table 14.1 extended with the interpretations of the consequence

p	q	r	$(p \rightarrow q)$	$(r \rightarrow p \vee q)$	$\boldsymbol{S_1}$	$(\mathrm{r} \rightarrow q)$	$\boldsymbol{S_2}$
T	T	T	T	T	**T**	T	**T**
T	T	F	T	T	**T**	T	**T**
T	F	T	F	T	**F**	F	**F**
T	F	F	F	T	**F**	T	**T**
F	T	T	T	T	**T**	T	**T**
F	T	F	T	T	**T**	T	**T**
F	F	T	T	F	**F**	F	**F**
F	F	F	T	T	**T**	T	**T**

The question we would like to answer is as follows: if John goes to the beach, is the weather nice? In other words, "Is $r \rightarrow q$ (S_2) a logical consequence of the premises in S_1?".

Table 14.1 shows the truth value of the premises for every possible interpretation (Step 1). Given that we have three propositions and two possible truth values, there are a total of $2^3 = 8$ interpretations.

Table 14.2 shows the truth table of the consequence we are looking for (Step 2), together with the truth table of the premises. According to the definition of logical consequence, S_2 is a logical consequence of S_1 ($S_1 \vDash S_2$), if all models of S_1 are also models of S_2. In other words, we need to look at the column of $S_1$and check if, for all T values, the column of S_2 also has a T value. Since this is the case for S_1 and S_2, we can say the S_2 is a logical consequence of S_1.

14.1.1.5 Proof Systems

Semantic methods for checking logical consequences (e.g., truth tables) are typically quite straightforward to understand as they directly work with the interpretations of sentences. Unfortunately, as the number of propositional constants of a language grows, so does the number of interpretations exponentially.[3] If the number of logical

[3]The problem is NP-complete. See also https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

constants in a propositional theory is large, the number of interpretations may be unfeasible to work with.[4] Deductive (proof) systems allow checking the logical consequence relationship via syntactic means. We can create a "proof" of a conclusion by manipulating a set of premises syntactically. In average, this requires much less effort than building a truth table for the premises and conclusion. If there is a proof of a formula f from a set of premises S using the rules of inference and axioms of a proof system, f is said to be *provable* from S, written as

$$S \vdash f$$

Note that unlike logical consequence ($\models$), provability ($\vdash$) is a syntactic operation, i.e., instead of checking the models of a set of formulas (e.g., with truth tables), we try to prove a conclusion by manipulating the symbols in the premises.

Before explaining the details of proof systems, we need to mention an important notion called *schema*. A schema is an expression that obeys the syntax rules of the language, in this case, the rules explained in Sect. 14.1.1.2, but uses metavariables in place of various components of the expression. They can be seen as formula patterns. For instance, the axiom schema called *implication reversal* using the metavariables ϕ and ψ can be written as follows:

$$(\neg\phi \Rightarrow \neg\psi) \Rightarrow (\psi \Rightarrow \phi)$$

An instance of a schema is the formula obtained by replacing the metavariables. For example, the following is an instance of the implication reversal schema:

$$(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$$

Note that metavariables can be replaced with any well-formed formula and not only with atomic formulas (e.g., propositional symbols).

The basis for proof systems is the use of axioms and rules of inference that can be applied to sentences to derive conclusions that are guaranteed to be correct under all interpretations that are a model of the original formulae.[5]

An *axiom* is a schema as described above.[6] Some valid axioms are as follows[7]:

- Reflexivity: $\varphi \Rightarrow \varphi$
- Negation elimination: $\neg\neg\varphi \Rightarrow \varphi$
- Negation introduction: $\varphi \Rightarrow \neg\neg\varphi$
- Tautology: $\varphi \vee \neg\varphi$

[4] Section 14.1.1.5 is mainly adapted from http://intrologic.stanford.edu/chapters/chapter_04.html.

[5] See also https://math.stackexchange.com/questions/1319338/difference-between-logical-axioms-and-rules-of-inference.

[6] Schemas are also sometimes called axiom schemas.

[7] Taken from http://intrologic.stanford.edu/chapters/chapter_04.html



Fig. 14.3 An application of modus ponens

An *inference rule* derives inferences from a set of formulas. It specifies a reasoning pattern consisting of some schemas as premises and another schema as conclusion. An example of this is *modus ponens* as follows:

$$\varphi \Rightarrow \psi$$
$$\varphi$$

$$\psi$$

A proof in a deductive system is a sequence of formulas where each formula is either a premise, is an axiom instance, or can be inferred from previous formulas in the sequence. We say that the sequence of formulas *proves* the last formula in the sequence.

At the beginning of this section, we claimed that proof systems provide an alternative way to check the logical consequence relationship. For this to happen, a proof system must have two important properties: soundness and completeness. A rule of inference (and a proof system) is *sound* if and only if, for every application of the inference rule, the premises logically entail the conclusions. In other words, if a formula is provable by means of syntactic manipulation, then it should also be a logical consequence. A rule of inference is *complete* if and only if it produces all formulas that are logical consequences of the premises. In other words, if a formula is a logical consequence of premises, then it can also be proven by the means of syntactic manipulation.

A proof system can be used to check the logical consequence relationship, only if it is sound and complete at the same time. This allows us to automate the determination of the logical consequence relationship.

The specific Hilbert system, which we will introduce, is a sound and complete proof system. It has modus ponens as the rule of inference and three axioms, namely, implication creation, implication distribution, and implication reversal.[8,9]

Modus ponens is a rule of inference that can be informally described as "*A* implies *B*," "*A* is true," and then "*B* must also be true." This inference is also called "affirming the antecedent" or "implication elimination."[10] Figure 14.3 shows an

[8] Adapted from http://intrologic.stanford.edu/chapters/chapter_04.html

[9] See also Ben-Ari (2012).

[10] https://en.wikipedia.org/wiki/Modus_ponens

1. $p \rightarrow q$	Premise
2. $q \rightarrow r$	Premise
3. $(q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$	Implication Creation
4. $p \rightarrow (q \rightarrow r)$	Modus Ponens: 2, 3
5. $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$	Implication Distribution
6. $(p \rightarrow q) \rightarrow (p \rightarrow r)$	Modus Ponens: 4, 5
7. $p \rightarrow r$	Modus Ponens: 1, 6

Fig. 14.4 A Hilbert proof

example application of the modus ponens rule. The first formula says, "If a student studies, then she passes the test." The second one says, "The student studies." Therefore, we can infer with modus ponens that "The student passes the test."

The *implication creation* axiom allows us to infer an implication from a formula in the proof. If we have ψ in the proof (e.g., proven at any step of the proof), then we can say that any φ implies ψ. Formally, the axiom can be written as $\psi \Rightarrow (\varphi \Rightarrow \psi)$ schematically.

The implication creation works because given ψ in the current set of formulas in a proof, adding $\psi \Rightarrow (\varphi \Rightarrow \psi)$ does not change the truth value of that set of formulas as the newly introduced formula is a tautology.

The *implication distribution* axiom specifies that an implication can be distributed over other implications. Formally, we can write this axiom as $(\psi \Rightarrow (\phi \Rightarrow \chi)) \Rightarrow ((\psi \Rightarrow \phi) \Rightarrow (\psi \Rightarrow \chi))$.

Implication reversal is an axiom that specifies a reversed implication of negated formulas can be inferred. This means that if $(\neg\phi \Rightarrow \neg\psi)$, then we can derive $(\psi \Rightarrow \phi)$.[11,12] This can then be written as $(\neg\phi \Rightarrow \neg\psi) \Rightarrow (\psi \Rightarrow \phi)$ schematically.

Figure 14.4 shows the proof of a conclusion $p \rightarrow r$ from a given set of premises $S = \{p \rightarrow q, q \rightarrow r\}$.[13] On the left-hand side of the figure, we see an inferred formula; on the right-hand side, we see via which rule or axiom the formula on that line is applied. Lines 1 and 2 show the two premises in our proof. The formula in Line 3 is an instance of the implication creation axiom after applying it on the premise in Line 2. Here, p is selected for the place of φ in the implication creation axiom. After adding Line 3 to the proof, we can apply modus ponens to the formulas in Lines 2 and 3 to obtain the formula in Line 4. The implication in Line 4 is distributed. Given that we have Line 4 in our proof, we can use the implication distribution axiom to infer the formula in Line 5. As we can see in Line 5, the formula in Line 4 implies $((p \rightarrow q) \rightarrow (p \rightarrow r))$. Given that we have Line 4 in our proof, the formula in Line 6 is implied. Finally, we prove $p \rightarrow r$ in Line 7 by applying modus ponens to the premise in Line 1 and the formula in Line 6.

[11] This axiom can be easily proven by writing both formulas as disjunctions.

[12] http://logica.stanford.edu/logica/homepage/hilbertarian.php?id=implication_reversal

[13] Adapted from http://intrologic.stanford.edu/chapters/chapter_04.html

$$
\begin{aligned}
&(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p) \\
&\qquad \equiv (\neg p \vee q) \rightarrow (\neg\neg q \vee \neg p) && \textit{(1)} \\
&\qquad \equiv \neg(\neg p \vee q) \vee (\neg\neg q \vee \neg p) && \textit{(2)} \\
&\qquad \equiv (\neg\neg p \wedge \neg q) \vee (\neg\neg q \vee \neg p) && (3) \\
&\qquad \equiv (p \wedge \neg q) \vee (q \vee \neg p) && \textit{(4)} \\
&\qquad \equiv (p \vee q \vee \neg p) \wedge (\neg q \vee q \vee \neg p) && \textbf{\textit{(5)}}
\end{aligned}
$$

Fig. 14.5 Converting a formula into the clausal form

14.1.1.6 Example: Resolution

Propositional resolution[14] is an important rule of inference for automated logical consequence checking (Bruni 2018). By using only one rule, it is possible to build a sound and complete theorem prover for all propositional logic with resolution.[15] Propositional resolution works only on expressions in *clausal form.* Fortunately, any logical formula can be turned into the clausal form via syntactic manipulations.

A *clause* is a set of literals which is assumed (implicitly) to be a disjunction of those literals. A literal is a propositional symbol or its negation. For instance, $\neg p \vee q \vee \neg r$ is a clause. Unit clauses are clauses with only one literal, e.g., $\{\neg q\}$. The clausal form of a formula is the implicit conjunction of clauses. For example, $p \wedge (\neg p \vee q \vee \neg r) \wedge (\neg q \vee q \vee \neg r) \wedge (\neg q \vee p)$ is a formula which can be written in the clausal form as $\{\{\neg p\}, \{\neg p, q, \neg r\}, \{\neg q, q, \neg r\}, \{\neg q, p\}\}$. The same clausal form can be written with an abbreviated notation as $\{p, \overline{p}q\overline{r}, \overline{q}q\overline{r}, \overline{q}p\}$.

Let us demonstrate the conversion to the clausal form with an example. Given the formula $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$, Figure 14.5 shows the steps on how to transform a formula into an equivalent formula in clausal form. The first step expands the implications on the left- and right-hand sides into disjunctions. Then in Step 3, the main implication is turned into a disjunction. In Step 4, the double negations are evaluated. Finally, in Step 5, the propositions are distributed over the disjunction operator to obtain the clausal form.

The proof system consists of one inference rule, that is, the resolution rule and a series of steps for applying this rule. Before we dive into the proof system based on resolution, we first introduce some notation:

- l is a literal (i.e., an atomic formula or its negation), and l^c is the complement of l (e.g., (l: p, l^c: $\neg p$), (l: $\neg q$, l^c: q)).
- C is a clause (a *set* of literals) (e.g., if p and q are literals, then *{p, q}* is a clause C which is a disjunction of those literals).
- S is a formula in clausal form – a *set* of clauses (e.g., $S = \{p, \overline{p}q\overline{r}, \overline{q}q\overline{r}, \overline{q}p\}$).

[14] Section 14.1.1.6 is largely based on Chap. 4 of Ben-Ari (2012).

[15] See also http://logic.stanford.edu/intrologic/notes/chapter_06.html.

Input: S_0 – a set of clauses that represent the premise and the negated conclusion
Output: "S_0 is satisfiable or "S_0 is not satisfiable". In the first we prove the conclusion as not a conclusion of the premises, the second case we prove the opposite.

1. Set S_0
2. To construct S_{i+1}, choose a pair of clashing literals and clauses C_{i1}, C_{i2} in S_i (if there are any) and derive:
   - $C_i := Resolve(C_{i1}, C_{i2})$
   - $S_{i+1} := S_i \cup \{C_i\}$
   - If C_i is the empty clause, output "S is not satisfiable", therefore the consequence is proven;
   - If $S_{i+1} = S_i$, output "S is satisfiable" and the consequence is not proven
3. Otherwise, set $i := i + 1$ and go back to Step 2.

Fig. 14.6 The resolution rule and its application

- Suppose C_1 and C_2 are clauses such that l is in C_1 and l^c is in C_2. The clauses C_1 and C_2 are said to be *clashing clauses*, and they clash on the complementing literals l and l^c.
- C is called the resolvent of C_1 and C_2, iff $C = Resolve(C_1, C_2) = (C_1 - \{l\}) \cup (C_2 - \{l^c\})$.
  - C_1 and C_2 are called the *parent clauses* of C.

In some cases, there can be multiple clashing literals in two clauses. In this case, only one pair of clashing literals can be resolved at a time. For example, the clauses $C_1 = \overline{p}q\overline{r}$ and $C_2 = \overline{q}p$ clash on $p, \overline{p}$ and $q, \overline{q}$. At one step, *only one pair of complementary literals* (I, I^c) can be selected (either (1) or (2)):

(1) $Resolve(C_1, C_2) = q\overline{r} \cup \overline{q} = q\overline{r}\,\overline{q}$
(2) $Resolve(C_1, C_2) = \overline{p}\,\overline{r} \cup p = \overline{p}\,\overline{r}p$

An important point of resolution is that it is a refutation system. This means that the proof of a conclusion (and due to soundness and completeness, logical consequence) is achieved via the refutation (proving unsatisfiability) of a set of formulas containing the negation of the conclusion. The intuition behind a refutation system is as follows. Assume that we have a set of formulas A and want to show that $A \vdash \alpha$ and, due to the soundness and completeness feature of the resolution rule, $A \vdash \alpha \leftrightarrow A \vDash \alpha$. Remember that the logical consequence relationship in this case means "every interpretation that makes A true also makes α true." This means there cannot be any interpretation that makes A true, which also makes $\neg\alpha$ true. Thus, whenever $A \vdash \alpha$, the set of formulas $A \cup \neg\alpha$ must be a contradiction (unsatisfiable) as there cannot be any interpretation that makes both A and $\neg\alpha$ true. This way, by proving the contradiction of set of formulas consisting of the union of A and $\neg\alpha$, we show that $A \vdash \alpha$.[16]

Figure 14.6 shows the resolution rule and how it is applied. Before we start with the resolution, we first need to convert the premises and the sought consequence into the clausal form. Assume that we have a set of formulas A and want to show that

[16] You can try this with a truth table consisting of truth values of A and α. If $A \vdash \alpha$ (thus $A \vDash \alpha$), then whenever A is true, α is also true. If we take $\neg\alpha$, then there is no row in the truth table where both A and $\neg\alpha$ are true. Therefore, we reach to a contradiction.

$$(p \rightarrow q) \wedge \neg(\neg q \rightarrow \neg p)) \equiv (\neg p \vee q) \wedge \neg q \wedge \neg\neg p$$
$$\equiv (\neg p \vee q) \wedge \neg q \wedge p$$

Fig. 14.7 Transformation into clausal form

$A \vdash \alpha$. To do that, we first add the negation of α ($\neg\alpha$) to A. Since the resolution is a refutation system, we try to prove the unsatisfiability of the obtained set of formulas ($A \cup \neg\alpha$). For the clausal form, we convert the obtained formula into a conjunction of disjunctions (a set of clauses), to obtain the set of clauses S_0, which is the input for our resolution process.

Starting from the initial set of clauses, the resolution rule iteratively builds a new set of clauses by adding the resolvent clause C_i to the current set of clauses, until a clause C_n that is an *empty clause* is achieved. When we reach to an empty clause, we have proved that $A \cup \neg\alpha$ is unsatisfiable (contradiction) and, therefore, $A \vdash \alpha$. The attempt to obtain the empty clause is a consequence of having the formulas in the clausal form. A clause is a disjunction of literals, and a set of clauses is a conjunction of clauses. Any time we have two clauses in a set consisting only of literals that are complement of each other, we obtain a contradiction (e.g., $p \wedge \neg p$). The resolution of these two complementing clauses gives us the empty clause.

Let us demonstrate the proof via resolution with an example. Given $(p \rightarrow q)$ as premise, let us show that $(\neg q \rightarrow \neg p)$ is provable. Since the resolution rule accepts a set of clauses, we first negate the sought conclusion and add to the premises as a conjunction. Now, we try to show that $(p \rightarrow q) \wedge \neg(\neg q \rightarrow \neg p)$ is unsatisfiable, i.e., a contradiction. First, we transform the formula into clausal form in Fig. 14.7.

Second, we start applying the resolution rule, i.e., show using resolution that $S_0 = \{\overline{p}q, \overline{q}, p\}$ is not satisfiable:

1. Let $S_0 = \{\overline{p}q, \overline{q}, p\}$.
2. The resolution rule says that we need to find two clauses that contain clashing literals (e.g., q and not q). In Step 2, these clauses are $C_{o1} = \overline{p}q$ and $C_{02} = \overline{q}$. These clauses clash on $l = q$ and $l^c = \overline{q}$. Therefore, the resolvent

$$C_0 = Resolve(C_{01}, C_{02}) = (\overline{p}q \backslash q) \cup (\overline{q} \backslash \overline{q}).$$
$$= (\overline{p}) \cup \emptyset$$
$$= \overline{p}$$
$$S_1 = S_0 \cup \{C_0\}$$
$$S_1 = \{\overline{p}q, \overline{q}, p, \overline{p}\}.$$

3. Note that in the first round, we neither reached a C that is an empty clause nor a state where $S_0 = S_1$. Therefore, we repeat Step 2 in this example for S_1. When we take the clauses $C_{11} = p$ and $C_{12} = \overline{p}$ from S_1 as parent clauses for resolution, we can derive the empty clause C_2 from it.

A derivation of the empty clause from S is called a refutation of S. This means we managed to show that given $(p \rightarrow q)$, we can prove $(\neg q \rightarrow \neg p)$.

14.1.1.7 Summary

We focused on three core aspects of propositional logic, namely, syntax, semantics, and reasoning. Syntax provides a vocabulary for expressing concepts without ambiguity. Semantics provides a connection to reasoning via *interpretations* and *models*. In other words, it defines what the syntax means. Reasoning deals with how to prove things. We provided a few proof systems for it.[17]

Propositional logic has certain limitations. Propositions are atomic, which means we have no access to their internal structure. "All humans are mortal" and "Socrates is a human" are two unrelated propositions. We cannot conclude "Socrates is mortal," without an explicit implication. There is no proper way to represent quantifications like "all" or "some." Let us try to express "All students know arithmetic." For this, very many propositions are needed:

$$DennisIsAStudent \rightarrow DennisKnowsArithmetic$$

$$JulietteIsAStudent \rightarrow JulietteKnowsArithmetic$$

$$\ldots$$

First-order logic, which we will introduce next, overcomes these limitations.

14.1.2 First-Order Logic

In this section, we will introduce first-order logic, particularly its motivation, syntax, and notions relevant to semantics and reasoning such as interpretations, model theory, and proof systems. Finally, we will provide a summary.[18]

14.1.2.1 Motivation

Assume we have the following propositional formulas:

- "Dennis is hardworking." (p)
- "Dennis is a student." (q)
- "Dennis knows arithmetic." (r)

[17] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[18] Section 14.1.2 is largely based on Ben-Ari (2012). See also (Fitting 1996) and (Huth and Ryan 2004).

- "*Dennis is hardworking*" $\wedge$ "*Dennis is a student.*" $\rightarrow$ "*Dennis knows arithmetic.* ($p \wedge q \rightarrow r$)

Given the premises p, q, and $p \wedge q \rightarrow r$, we can apply modus ponens as follows to infer "Dennis knows arithmetic."

$$
\begin{array}{l}
p \\
q \\
p \wedge q \rightarrow r \\
\text{----------} \\
r
\end{array}
$$

But now, suppose we want to infer "Juliette knows arithmetic." Nothing we have stated so far will help us. We would need to add three more premises about Juliette akin to the ones about Dennis to infer "Juliette knows arithmetic." The problem is that we cannot represent any of the details of these propositions. It is the *internal structure* of these propositions that makes the reasoning valid.

In propositional logic, we do not have anything else to talk about besides propositions, and we cannot access their internal structure. We cannot express that *anyone* who is a hardworking student knows arithmetic. A more expressive logic is needed to express richer things (talking about a set of objects and not only about singular objects). *First-order logic (FOL)* is such a logic that allows us to overcome such limitations of propositional logic.

14.1.2.2 Syntax

The syntax of FOL consists of the following elements:

- Constants (null-ary functions) are names of specific objects, e.g., *harry*, *albus*, *ginny*, and *32*.
- Functions map objects to objects, e.g., *father*(*harry*), *age*(*ronald*), and *max*(*25*, *42*).
- Variables are used for statements about unidentified objects or general statements, e.g., x, y, z, etc.
- A term can be a constant, variable, or function with n arguments, where each argument is a term. Terms without variables are called *ground terms*, for example, c, $f(c)$, $g(x, x)$, g($f(c)$), and $g(x, y)$.
- Predicate symbols represent relations between zero or more objects. A predicate's arity defines the number of objects it uses. Examples: *likes*(*harry, ginny*), *likes*(*x, x*), *friends*(*fatherof*(*rose*), *fatherof*(*albus*)).

The set of constants, function, and predicate symbols that are used to build a set of formulas is called a *signature*.

Every skyscraper is a building:
$\forall x\big(skyscraper(x) \rightarrow building(x)\big)$

Some buildings are skyscrapers:
$\exists x\big(building(x) \rightarrow skyscraper(x)\big)$

Every skyscraper is bought by some organization or a person:
$\forall x(skyscraper(x) \rightarrow \exists y((organization\,(y) \vee person(y)) \wedge boughtBy(x, y))$

Fig. 14.8 Example first-order formulas

FOL formulas are joined together by *logical operators* to form more complex formulas (just like in propositional logic). The basic logical operators are the same as in propositional logic, such as negation, conjunction, disjunction, implication, and equivalence. Additionally, FOL defines two new quantifiers, namely, universal *(∀)* and existential *(∃)* quantifiers. The quantifiers allow us to express properties of collections of objects instead of enumerating objects by name. Universally quantified variables make their predicates true for *all* substitutions of the variable. Existentially quantified variables make their predicates true for *at least one* substitution of the variable.

A formula is defined by the following rules by induction:[19]

- *Predicate symbols*: If p is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms, then $p(t_1,\ldots,t_n)$ is a formula.
- *Negation*: If ϕ is a formula, then $\neg\phi$ is a formula.
- *Binary connectives*: If ϕ and ψ are formulas, then any formula with a binary logical connective connecting ϕ and ψ is a formula.
- *Quantifiers*: If ϕ is a formula and x is a variable, then $\forall x\,\phi$ and $\exists x\,\phi$ are formulas.

Atomic formulas are formulas obtained only using the first rule. Any occurrence of a variable in a formula *not* in the scope of a quantifier is said to be a *free* occurrence. Otherwise, it is called a *bound* occurrence. Thus, if x is a free variable in ϕ, it is bound in $\forall x\,\phi$ or $\exists x\,\phi$. A formula with no free variables is called a *closed formula*. We consider, in our context, only closed formulas. Figure 14.8 shows example formulas in FOL syntax.

14.1.2.3 Interpretation

Interpretations in FOL map symbols of a logical theory (constants, functions, predicates) onto objects, functions, and relations of the domain of discourse or universe.[20] A valuation function assigns domain objects to variables.

[19] See also https://en.wikipedia.org/wiki/First-order_logic#Syntax.

[20] See also https://en.wikipedia.org/wiki/First-order_logic#Semantics.

A *domain* is a set of objects D, where $d_1 \ldots d_{|D|} \in D$. F denotes functions over D with arity n where F: $D^n \rightarrow D$. R denotes relations over D where $R \subseteq D^n$. First, basic terms can be interpreted with a mapping I as follows, given c, f, and P are constant, function, and predicate symbols, respectively:

- Constants: $I(c) = d$ (an object in the domain).
- Functions: $I(f) = F$ (a function over the domain).
- Predicates: $I(P) = R$ (a relation over the domain).

Next, we build the semantics for complex terms and formulas constructively, based on the basic interpretation mapping and the valuation function V, where $V(x) = d$ and x is a variable, d is an object in the domain, and α and β are first-order formulas.

- Terms with variables: $I\,(f(t_1,...,t_n)) = I\,(f)\,(I\,(t_1),...,\, I\,(t_n)) = F(I\,(t_1),...,\, I\,(t_n)) \in D$ where $I(t_i) = V(t_i)$ if t_i is a variable.
- Atomic formulas: $I\,(P(t_1,...,t_n))$ is true if $(I\,(t_1),...,\, I\,(t_n)) \in I\,(P) = R$.
- Negated formulas: $I\,(\neg\alpha)$ is true if $I\,(\alpha)$ is not true.
- Complex formulas:
  - $I(\alpha \vee \beta)$ is true if $I\,(\alpha)$ or $I\,(\beta)$ is true.
  - $I\,(\alpha \wedge \beta)$ is true if I (α) and $I\,(\beta)$ are true
  - $I\,(\alpha \rightarrow \beta)$ if I (α) is not true or $I\,(\beta)$ is true.

  Quantified formulas:

- $I\,(\exists x{:}\alpha)$ is true if α is true with $V\,(x) = d$ for some $d \in D$.
- $I\,(\forall x{:}\alpha)$ is true if α is true with $V(x) = d$ for all $d \in D$.

Note that for quantified formulas, the order of the quantification makes an important difference in how the formulas are interpreted. $\forall x \exists y{:}\alpha$ means that for all values of x, there must be at least one object that makes predicates with y true. $\exists y \forall x{:}\alpha$ means there must be at least one object making the predicates with y true, for all values of the x.

14.1.2.4 Model Theory

Like in propositional logic, the model theory of FOL deals with the logical consequence relationship (⊨). Given an interpretation I into a domain D with a valuation V and a formula ϕ, we say that ϕ is *satisfied* in this interpretation or that this interpretation is a *model* of ϕ iff $I(\phi)$ is true. This means the interpretation I into a domain D (with valuation V) holds true for the formula ϕ.

Given a set of formulas F and a formula α, α is a *logical consequence* of F if and only if α is true in every interpretation in which F is true ($F \models \alpha$). This means every model of F is also a model of α.

14.1.2.5 Proof Systems

Like propositional logic, sound and complete proof systems enable automation for finding the logical consequence relationship. Inference rules like modus ponens and resolution for propositional logic apply to predicate logic as well (Dyer 1998; Brachman and Levesque 2004). Therefore, we will not cover them here in detail. However, there are new inference rules that are suitable for quantified formulas to be used in proof processes. These are, namely, universal elimination, existential elimination, existential introduction, and generalized modus ponens (see (Dyer 1998) for the definitions of these rules[21]).

Universal elimination allows replacing a universally quantified variable with any constant that is denoting an object in the domain. If $\forall x\ P(x)$ is true, then $P(c)$ is true, where c is *any* constant symbol interpreted by $I(c)$. Let us take the following formula as an example: $\forall x StudiesFor(x, lecture) \rightarrow Passes(x, lecture)$. Assuming a constant named *dennis* is in the domain, we can infer the ground formula $StudiesFor(dennis, lecture) \rightarrow Pass(dennis, lecture)$.

Existential elimination allows the replacement of a variable with a constant that did not so far appear in the set of formulas. If $\exists x\ P(x)$ is true, then we can infer $P(c)$. This process is also called *skolemization* and c is a Skolem constant. Skolemization is particularly useful with inference rules like resolution. As we have seen before, resolution requires a clausal form, which requires formulas to only have universally quantified variables. Let us take the formula $\exists x\ StudiesFor(x)$. Assuming that *someLecture* is a new constant, we can conclude $StudiesFor(someLecture)$.

Existential introduction is a rule that is the inverse of existential elimination. If $P(c)$ is true, then $\exists x\ P(x)$ is inferred. By applying this rule, we can replace all occurrences of a given constant symbol with a new variable symbol that does not exist anywhere else in the formula. For example, if we know $StudiesFor(dennis, lecture)$ as true, we can infer that $\exists x\ StudiesFor(dennis, x)$.

Generalized modus ponens (GMP) is an inference rule that applies modus ponens reasoning to generalized formulas. It is typically combined with an application of universal elimination. For example, given $P(a)$, $Q(a)$, and $\forall x\ (P(x) \wedge Q(x) \rightarrow R(x))$, we can derive $R(a)$. GMP requires substitutions for variable symbols. Assume that $sub(\theta, \alpha)$ represents a set of substitutions θ to the formula α. $\theta = \{v_1/t_1, v_2/t_2, ..., v_n/t_n\}$ means to replace all occurrences of variables $v_{1 \ldots} v_n$ by terms $t_{1 \ldots} t_n$. The example in Fig. 14.9 shows two ground formulas and a formula in the form of a Horn clause. With the substitution $\theta = \{x/Kevin, y/Umut, z/Elwin\}$, we can infer $Faster(Kevin, Elwin)$.

[21] See also http://intrologic.stanford.edu/chapters/chapter_10.html.

$$Faster(Kevin, Umut),$$
$$Faster(Umut, Elwin)$$
$$Faster(x, y) \wedge Faster(y, z) \rightarrow Faster(x, z)$$

$$Faster(Kevin. Elwin)$$

Fig. 14.9 An example application of GMP

14.1.2.6 Summary

In difference to propositional logic, general FOL is no longer decidable. However, FOL is still *semi-decidable*. There are sound and complete proof methods like resolution, but this does not imply a decision process always terminates.

A theory is semi-decidable if there is an effective method that, given an arbitrary formula, will always tell correctly when the formula is in the theory, but may give either a negative answer or no answer at all when the formula is not in the theory.[22] For example, the resolution rule may not terminate for a FOL proof.

In mathematical logic, *Lindström's theorem*[23] states that FOL is the *strongest logic* where a countable set of first-order sentences has a model if and only if every finite subset of it has a model.

Spoken in simpler terms, FOL is the *strongest* logic that can be meaningfully handled by computational logic (however semi-decidability only). That is, we often may not know whether we stop the proof too early. However, we know that if a set of formulas has an infinite model, then it also must have a finite one that can be found by finite enumeration. Second-order logic adds significant expressive power to FOL but is not manageable by approaches of computational logic. FOL is *computationally hard* and *only semi-decidable.*

As the "strongest" logic, there are still certain disadvantages of FOL. FOL can express neither the deductive closure of a set of formulas nor the transitive closure of a relationship. For example, the logical theory $P(a, b)$, $P(a, c)$ has the following models:

- *{P'(a', b'), P'(a', c')}*
- *{P'(a', a'),P'(a', b'), P'(a', c')}*
- ....
- *{P'(a', a'),P'(a', b'), P'(a', c'), P(b', a'),P'(b', b'), P'(b', c'), P'(c', a'),P'(c', b'), P'(c', c')}*

That is, FOL cannot exclude facts from being true that intuitively do not follow from this theory. Apart from the issues with deductive and transitive closure, FOL also has limitations in terms of the statements it can make. First order means that statements can be only done about elements of sets and not about sets. This means

[22] https://en.wikipedia.org/wiki/Decidability_(logic)

[23] https://en.wikipedia.org/wiki/Lindstr%C3%B6m%27s_theorem

we can talk about an object that is a member of a predicate; however, we cannot talk about that predicate itself. To overcome these limitations, there are certain syntactical modifications of FOL that simulate second-order metamodelling features. We have already seen some of those languages in the previous chapters, such as F-logic. Naturally, such statements about statements can be natively done with higher-order logical languages, which we will very briefly cover at the end of Sect. 14.1.

There are restricted formalisms that make FOL more tractable or even decidable. An example is *monadic FOL*.[24] Another example is called *description logic* which we will discuss next. Moreover, there are non-standard model theories, such as *minimal model semantics* that extend the expressive power of a restricted version of FOL. We will cover minimal model semantics later in Sect. 14.1.4.

14.1.3 Description Logic

We introduce description logic (DL), a restricted version of FOL that enables decidable reasoning.[25] We first give a motivation for such a logical formalism and then introduce its syntax. We cover different subsets of the DL language family, starting with the most compact one called ALC, and then we mention the others briefly that are built on top of ALC. After that, we cover the semantics as we did with propositional and first-order logic and introduce interpretations, model theory, and proof systems for DL. Finally, we give a summary.

14.1.3.1 Motivation

As we have already discussed, entailment in FOL is only semi-decidable. This means that a proof process may not terminate if there is no proof. From a practical point of view, this is problematic as we cannot distinguish between the non-existence of a proof and the failure of an implementation to find one in a reasonable time. The completeness of a proof system does not help too much either since proofs can still be intractable. So, in addition to decidability, identifying tractable subsets of the logical formalism and the supported reasoning tasks are an important issue, too.

For these reasons, often less expressive but decidable formalisms are more suitable. Description logic (DL) is one of these formalisms. It is logic that is the fundament of the *Web Ontology Language* (*OWL*). There are different DLs that

[24] Monadic first-order logic is the fragment of first-order logic in which all relation symbols in the signature are monadic (i.e., they take only one argument) and there are no function symbols: https://en.wikipedia.org/wiki/Monadic_predicate_calculus.

[25] Section 14.1.3 is largely based on Baader et al. (2003), Baader (2009), Hitzler et al. (2009), and Rudolph (2011).

correspond to different fragments of FOL. Most DLs are based on a two-variable fragment of FOL. A DL defines classes (concepts) that correspond to unary predicates and properties (roles) that correspond to binary predicates.

There are also other restrictions, such as not having function symbols, for the sake of decidability. Most DLs are decidable and usually have decision procedures for key reasoning tasks.

14.1.3.2 Syntax

Description logic provides a large syntax with the following major components:

- Concepts/classes: e.g., Person, Female, etc.
- Top and bottom concepts: $\top$ and $\bot$
- Roles: e.g., hasChild
- Individuals: e.g., Mary and John
- Constructors:
  - Union $\sqcup$: e.g., $Man \sqcup Woman$
  - Intersection $\sqcap$: e.g., $Engineer \sqcap Mother$
  - Existential restriction $\exists$: e.g., $\exists knows.\ Lawyer$ (knows some lawyer)
  - Universal restriction $\forall$: e.g., $\forall hasChild.\ Person$ (all children someone has are people)
  - Complement/negation $\neg$: e.g., $\neg Mother$
  - Cardinality restriction $\geq_n$, $\leq_n$: e.g., $\leq_1 hasSpouse$
  - Axioms: e.g., subsumption $\sqsubseteq$: $Mother \sqsubseteq Parent$

Classes/concepts are sets of individuals. We can distinguish different types of concepts. Atomic concepts cannot be further decomposed. For example, *Person*, *Organization*, *Man*, and *Woman* can all be atomic concepts. A concept can be defined *partially* or *completely*. A partial concept is defined via a subsumption axiom. For example, $Man \sqsubseteq Person$ has the intended meaning that if an individual is a man, we can conclude that it is a person. A complete concept definition is done via an equality axiom. For example,

$$Man \equiv Person \sqcap Male$$

has the intended meaning that every individual who is a *Man* is a *Person* and a *Male* and vice versa. There are two special kinds of built-in concepts in DL, namely, the top ($\top$) and bottom ($\bot$) concepts. The top concept is the super concept of every concept which means all individuals are an instance of this concept. Contrarily, the bottom concept is a concept that could not have instances.

Roles relate individuals to each other, e.g., *directedBy*(*1917*, *SamMendes*) and *hasChild*(*harry*, *albus*). Roles can have a domain and range. For example, $\exists directedBy.\ \top \sqsubseteq Movie$ describes the domain of the role *directedBy as Movie*, and $\top \sqsubseteq \forall\ directedBy.\ Person$ describes its range as *Person*. Given those

definitions, we can conclude that *1917* is a movie and *Sam Mendes* is a person. There are different types of roles with different purposes. Functional roles have at most one unique value for an instance (semantically corresponds to partial functions). It is a special case of maximum cardinality restriction, e.g., $Person \sqsubseteq \leq_1 hasBirthMother.Mother$ says that every person has a maximum of one birth mother, which would imply that a *Person* instance has only one unique value[26] for the *hasBirthMother* property (see also Rudolph 2011).[27]

Transitive roles describe transitive relationships like being part of, ancestry, and so on. For example, $hasAncestor \circ hasAncestor \sqsubseteq hasAncestor$ says that *hasAncestor* is a transitive property.

Symmetric roles are roles that can connect individuals in a symmetrical relationship. For example, $hasSpouse \equiv hasSpouse^-$ says that if *A* has spouse *B*, then *B* has spouse *A*. Inverse roles specify the inverse of a relationship between two individuals. For example, $hasParent \equiv hasChild^-$ says that if *hasParent(A, B)*, then *hasChild(B, A)* and vice versa.

Typically, a DL knowledge base consists of a set of axioms. These axioms are split into two categories, namely, *TBox* and *ABox*.

A TBox (terminology) is a set of inclusion/equivalence axioms denoting the conceptual schema/vocabulary of a domain. For example,

$$Bear \sqsubseteq Animal \sqcap Large$$

$$hasParent \equiv hasChild^-$$

An ABox (assertions) is a set of axioms, which describe concrete instance data and make assertions about individuals. For example,

$$hasParent(charles, elizabeth)$$

$$Bear(YogiBear)$$

14.1.3.3 Attributive (Concept) Language with Complements

A core DL variant is the *Attributive (Concept) Language with Complements* (ALC), in the sense that many other DL variants are defined as extensions of this variant. ALC is the smallest propositionally closed description logic. Propositionally closed means that it provides (implicitly or explicitly) mechanisms for conjunction, union, and negation of class descriptions.

[26] Here, also an additional range restriction is specified with the Mother concept.

[27] In case there are multiple values for a functional property, these values are inferred as equivalent instances. If they are explicitly specified as different individuals, then the knowledge base is inconsistent.

C, D (*atomic concept*)
R (*role*)
$\top$ (*universal concept – superclass of all classes*)
$\bot$ (*bottom concept*)
$C \sqcap D$ (*intersection*)
$C \sqcup D$ (*disjunction*)
$\neg C$ (*negation*)
$\forall R.C$ (*value restriction via universal quantification*)
$\exists R.C$ (*existential quantification*)
$C \sqsubseteq D$ (*subsumption*)
$C \equiv D$ (*equality, implicitly* $C \sqsubseteq D$ *and* $D \sqsubseteq C$)
$C(a)$ *or* $a \in C$ (individual assertion)
$R(a, b)$ *or* $(a, b) \in R$ (*role assertion*)

Fig. 14.10 The modelling primitives supported by ALC

TBox:

$Person \sqcap \forall knows.Lawyer$	(1)
$Dog \sqsubseteq Animal$	(2)
$Man \equiv Male \sqcap Person$	(3)
$Person \sqcap \exists teaches.Lecture$	(4)

ABox:

$Person(Snape)$	(5)
$teaches(Snape, IntroductionToPotions)$	(6)

Fig. 14.11 A DL knowledge base defined with the ALC variant of description logic

ALC supports concepts, atomic roles, concept construction with the conjunction, disjunction and negation operators, as well as existential and universal quantifiers restricted to a specific class. Figure 14.10 shows a summary of the supported DL primitives in ALC.

Figure 14.11 shows a DL knowledge base defined with ALC. The first three axioms are part of the TBox. The first statement (1) defines an anonymous concept for "Persons who know only lawyers." The second statement (2) defines that every *Dog* instance is an *Animal* instance. The third statement (3) specifies that every *Man* instance is an instance of both *Male* and *Person* and vice versa. The fourth statement (4) defines an anonymous concept for "Persons who teach some lectures." The ABox contains two facts. The fifth statement (5) tells that Snape is an individual of concept Person. The sixth statement (6) says Snape teaches Introduction to Potions; in a more formal way, the pair (*Snape*, *IntroductionToPotions*) is an individual of role *teaches*.

14.1.3.4 The Holy Language Family

Description logic is a family of related logics. The members of this family have differences in expressivity and features, as well as the complexity of inference.

Description logics follow a naming schema according to their features. Some widely used examples are as follows[28]:

- **S** typically refers to *ALC* extended with transitive roles.
- **H** refers to role hierarchy.
- **O** refers to nominals – classes that are defined by their enumerated members (e.g., days of the week).
- **I** refers to inverse roles.
- **N** refers to unqualified number (cardinality) restrictions.
- **Q** refers to qualified number (cardinality) restrictions, where not only the number of values but also the range of a property is restricted. For example, a pizza must have at least two components, one of which is pizza base.
- **F** refers to functional properties.
- **R** refers to limited complex role inclusion axioms, role disjointness and other role axiom support like reflexivity and asymmetry.
- **D** refers to datatype support.

A good demonstration of how different DL variants are used in practice is the development of the Web Ontology Language (OWL).[29] DL is the underlying logical formalism. OWL 1.0 DL is based on the SHOIN(D) variant, which means that it supports ALC with transitive roles, role hierarchy, nominal classes, inverse roles, and cardinality restrictions, as well as datatype support. Meanwhile, OWL2 DL is based on the SROIQ(D), which means that it supports *ALC* with transitive roles, complex role axioms, nominal classes, inverse roles, and qualified cardinality restrictions.[30]

14.1.3.5 Interpretation, Model Theory, and Proof Systems

In this section, we cover how the symbols of ALC are connected to their meanings via interpretations. This section is mainly based on Rudolph (2011) and Baader (2009); therefore, we refer the readers there for more details.

Given a vocabulary, a set of names for concepts, roles, and individuals, an interpretation I is a pair $(\Delta^I, \cdot^I)$, where a non-empty set Δ^I is the domain and $\cdot^I$ is a mapping that maps:

- Names of individuals to elements of the domain for each individual name a, $a^I \in \Delta^I$
- Names of classes/concepts to subsets of the domain – for each concept name C, $C^I \subseteq \Delta^I$

[28] See also https://en.wikipedia.org/wiki/Description_logic#Naming_convention.

[29] See also Sect. 13.3.

[30] https://www.w3.org/2007/OWL/wiki/SROIQ

- Names of properties/roles to subsets of a set of pairs from the domain – for each role name R, $R^I \subseteq \Delta^I \, x \, \Delta^I$

The interpretations for more complex axioms can be built compositionally on the basic interpretations above:

- Bottom concept – $\bot = \emptyset$.
- Top concept – $\top = \Delta^I$.
- Negation – $(\neg C)^I = \Delta^I \setminus C^I$.
- Intersection – $(C \sqcap D)^I = C^I \cap D^I$.
- Disjunction – $(C \sqcup D)^I = C^I \cup D^I$.
- Existential quantification – $(\exists R.\ C)^I = \{a \in \Delta^I \mid \exists\ b \in \Delta^I \textit{ where } (a,b) \in R^I \textit{ and } b \in C^I\}$, i.e., the set of instances in the domain that fulfils the following condition: for an instance a in the domain, there exists an instance b in the domain where there is a pair *(a,b)* for which the relation R holds and b is an instance of concept C.
- Universal quantification – $(\forall R.\ C)^I = \{a \in \Delta^I \mid (a,b) \in R^I \textit{ implies } b \in C^I\}$, i.e., the set of instances in the domain that fulfils the following condition: whenever there is an instance b, connected with instance a with the relation R, b is an instance of concept C.

Like FOL, the model theory deals with the logical consequence relationship. In terms of DL, a formula a is a logical consequence of a knowledge base, if every model of the knowledge base is also a model of a. An interpretation is a model if it satisfies all formulas in the knowledge base given the above semantics via interpretations.

Alternatively, DL axioms can be mapped to the relevant subset of first-order logic (at least for the mainstream DL variants), and the interpretations and model theory of FOL can be used. This translation is usually very straightforward. Some basic transformations are as follows:

- A concept is converted to a first-order formula with one variable (e.g., the concept C is mapped to predicate *C(x)*, where x is a first-order variable).
- A role is converted to a first-order formula with two variables (e.g., the role R is mapped to a predicate *R(x,y)*, where x and y are first-order variables).
- The instantiation of a concept or a role is done via grounded predicates.
- More complex transformations can be done in a similar way, where, for example, concept unions are mapped to a disjunction of predicates and intersections are mapped to the conjunction of predicates. See Rudolph (2011) for more details and formal definitions of DL to FOL transformation.

The DL community defined various reasoning tasks, such as (Baader 2009):

- *Satisfiability*: Check if the assertions in a DL knowledge base have a model.
- *Instance checking*: Check if an instance is an element of a given concept.
- *Concept satisfiability*: Check if the definition of a concept is satisfiable.
- *Subsumption*: Check if concept B is a subset of concept A (i.e., if every individual of concept B is also an individual of concept A).

- *Equivalence*: Check if a concept A subsumes a concept B and B subsumes A.
- *Retrieval:* Retrieve a set of instances of a given concept.

Although the tasks appear different, they can be reduced to each other in polynomial time, which reduces the implementation effort for reasoners. See Baader (2009) for a detailed discussion.

Since the mainstream DLs can be mapped to FOL, the sound and complete proof systems, such as resolution, can be also used with DL knowledge bases to automate the reasoning tasks. There is, however, a particularly popular inference method called tableaux that is used by many DL reasoner implementations. Like resolution, tableaux algorithm is a refutation system, meaning the consequence is proven by finding a contradiction in the knowledge base when the negation of the sought consequence is added to the knowledge base. The tableaux method works with expansion rules in a knowledge base. These rules expand the knowledge base by finding explicit facts. The method looks for a contradiction in the expanded knowledge base to refute the negated formula. The tableaux method is sound and complete; however, it is typically quite expensive and may go up to exponential time complexity if the TBox contains cycles. See Horrocks (2005) and Horrocks and Sattler (2007) for the details of how the algorithm works.

14.1.3.6 Summary

In this section, we introduced a subset of FOL called description logic that sacrifices expressivity for more favorable computational properties. DL is a family of languages, and each member has a different set of modelling primitives. Like F-logic that we covered in Sect. 13, DL also contains modelling primitives like concepts and roles, which merge the epistemological layer with the logical layer. DL has decidable reasoning and sound and complete proof systems such as tableaux. Although it may be expensive, reasoning with this method always guarantees to terminate.

As the underlying formalism of OWL for the Semantic Web, DL adopts a standard model-theoretic FOL semantics which makes use of the open-world assumption. In the following section, we will cover another modification of FOL with modified semantics using the closed-world assumption, which is the core of logic programming.

14.1.4 Herbrand Model Semantics

The standard FOL semantics has an important feature: Symbols and their meaning are decoupled and must be connected via interpretations w.r.t. a specific domain. This domain can be an arbitrary set of constants, functions, and relations, i.e., we have an uncountable, heterogeneous collection of interpretations. A simplification for this is provided by Herbrand interpretations (Eiter and Pichler 2010a). Basically,

constants, functions, and relations are interpreted by themselves which relates to the unique name assumption. This also makes it possible to easily compare different Herbrand models of a logical theory. For example, without negation and disjunction, there exists a unique minimal model that can be taken as standard semantics and implements the closed-world assumption. There are also restricted ways to use negation (i.e., stratification can still be handled by a selection operator over several minimal models).

In this section,[31] we first talk about Herbrand models, an alternative way of defining interpretation and model theory for FOL. Second, we talk about minimal model semantics, which allows the definition of a minimal model of a logical theory. Such a definition is crucial for many real-world applications, and it is the core of deductive databases and logical programming. We then cover the perfect model semantics, which provides a mechanism for selecting a minimal model when there are multiple of them. Finally, we conclude with a summary.

14.1.4.1 Herbrand Models

Herbrand semantics for FOL also defines its model theory in terms of interpretations and models; however, the definition is different from the standard FOL definitions. Assume the following vocabulary:

- Constant symbols: a, b
- Function symbols: f
- Predicates: P, Q

The *Herbrand universe U* is the set of all ground terms that can be obtained from the constants and function symbols in a vocabulary. For example, $U = \{a', b', c', f'(a'), f'(f'(a')), f'(b')\ldots\}$.

A *Herbrand base B* is the set of all ground atoms that can be obtained from the predicate symbols and ground terms in the vocabulary, e.g., $B = \{P'(a'), Q'(b'), Q'(f'(a'))\ldots\}$.

A *Herbrand interpretation I* is a subset of the Herbrand base *B*. The domain of a Herbrand interpretation is the Herbrand universe U. The constants are interpreted by themselves. Every function symbol is interpreted by the function that applied to it. If f is an n-ary function symbol ($n>0$), then the mapping from U^n to U defined by

$$f(t_1', \ldots, t_n') \rightarrow f'(t_1', \ldots, t_n')$$

is assigned to f. For example, $I_1 = \{P'(a'), P'(f'(a'))\}$.

A *Herbrand model M* is a Herbrand interpretation which makes every formula in a set of formulas true:

[31] The content of this section is mainly based on Schöning (2008), Bachmair and Ganzinger (2001), and Eiter and Pichler (2010a).

$$\mathrm{S} = \{\mathrm{P(a)}, P(x) \rightarrow Q(x)\} \text{ then } I_2 = \{P'(a'), Q'(a')\} \text{ is a model of S.}$$

Let us remember how a logical consequence relationship is defined generally. B is a logical consequence of A if every model of the set of formula A is also a model of B. In standard FOL, this is a semantic operation; however, Herbrand interpretations are symbolic. There is no external world; symbols are mapped to themselves. For Herbrand models, we can define logical consequence as follows: B is a logical consequence of A, if any subset of the Herbrand base that satisfies A (Herbrand model of A) also satisfies B (is also a Herbrand model of B).

A short but effective example to show the difference between standard FOL semantics and Herbrand semantics is as follows. Consider a signature with just a single constant a and a unary predicate P. $U = \{a'\}, B = \{P'(a')\}$. Now, consider the set of formulas $S = \{P(a), \exists x \neg P(x)\}$. S is satisfiable in the standard first-order logic semantics, e.g., $\{P'(a'), P'(b')\}$ would be a model. However, under Herbrand semantics, S has no model, because there is no subset of B that satisfies S (Eiter and Pichler 2010a).

14.1.4.2 Minimal Model Semantics

Consider the following vocabulary and formula[32]:

- Constants: *Jeremiah, Kermit*
- Predicates: *Green(x), Frog(x)*

Consider the formulas $Green(Jeremiah)$ and $Green(x) \rightarrow Frog(x)$. There are many possible Herbrand interpretations.

$$I_1 = \{Green'(Jeremiah'), Green'(Kermit'), Frog'(Jeremiah'), Frog'(Kermit')\}$$

$$I_2 = \{Green'(Jeremiah'), Green'(Kermit'), Frog'(Jeremiah')\}$$

$$I_3 = \{Green'(Jeremiah'), Frog'(Kermit')\}$$

$$I_4 = \{Green'(Jeremiah'), Frog'(Jeremiah')\}$$

$$I_5 = \ldots.$$

Some of these interpretations are models; in fact, all of them containing $Green'(Jeremiah')$ *and* $Frog'(Jeremiah')$, i.e., there are three such models.

$$\mathrm{M}_1 = \{Green'(Jeremiah'), Frog'(Jeremiah')\}$$

[32] Adapted from Balke and Kroll (2020a)

$$\mathrm{M}_2 = \{Green'(Jeremiah'), Frog'(Jeremiah'), Frog'(Kermit')\}$$

$$\mathrm{M}_3 = \{Green'(Jeremiah'), Frog'(Jeremiah'), Green'(Kermit'), Frog'(Kermit')\}$$

In a query-answering scenario, we ideally want only one of them, which is *the minimal model.*

A given Herbrand model M_1 is minimal if and only if there is no other Herbrand model M_2 such that $M_2 \subset M_1$. When the minimal model is unique, then this model is called the *least* Herbrand model. It is the intersection of all Herbrand models. The (unique) minimal Herbrand model in our example is M_1. Minimal model semantics is the semantics adopted by logic programming languages. Datalog (with negation) is such a language.

There are several methods for finding the minimal model such as the fixed-point iteration. The fixed-point iteration approach works like tableaux in DL in the sense that the set of ground terms is expanded based on the existing ground terms at any given stage until no new ground term can be inferred. However, both with and without negation, the efficiency of the naïve fixed-point iteration approach is not great. At each step, we are recalculating everything from the previous step. Overall, we are deducing everything that can be deduced. We find all models of a set of sentences, regardless of whether it is relevant to our query. Then look for our sought consequence if it exists in the found models. In other words, the entire Herbrand base is built first; then a subset is searched (Balke and Kroll 2020a).

There are several optimization techniques that, e.g., focus on a logical rewriting of sentences to prune a number of steps taken in calculating the minimal models. A very popular one of them is the *magic sets technique* (Balke and Kroll 2020b). Magic sets (Bancilhon et al. 1985) is a query-rewriting method that reduces the number of rules considered while deducing a goal (answering a query) by calculating only relevant facts to that goal. The main idea is that given a program (a set of formulas) P, transform it to a Program P' that is semantically equivalent to P but can be evaluated faster using the same evaluation technique. The main disadvantage of this approach is that the logic program becomes rewritten (compiled) and this happens for every query. First, this is an additional computational overhead. Second, it may no longer be intuitively understandable. Its conclusions and runtime evaluation become non-understandable for humans. The whole idea of logic becomes spoiled.

When disjunction is involved, there can be multiple minimal models. Consider the following simple logical formula, $p(a) \vee q(a)$, which has two minimal models: $\{p'(a')\}$ *and* $\{q'(a')\}$. Any subset of it would no longer be a model. Both satisfy the program, but their intersection is empty, which means there is no unique minimal model. We need a deterministic criterion for deciding on the appropriate model (more on this in the next section) (Balke and Kroll 2020a).

A way to ensure the existence of a least minimal model is to restrict first-order logic to Horn logic without negation. In Horn logic, the formulas are restricted to Horn clauses (remember Horn clauses from Part I). A Horn clause is a formula that looks like a rule (IF .. Then ..). Horn clauses are implicitly universally quantified.

Remember that a Horn clause is a disjunction of literals with at most one positive literal, $\neg p \vee \neg q \vee ... \vee \neg t \vee u$, which can be written in implication form as $u \leftarrow p \wedge q \wedge ... \wedge t$.

14.1.4.3 Perfect Model Semantics

We have seen so far that a unique minimal model is possible if we forbid negation and disjunction in the premise. What happens if we allow Horn logic with negation? Consider the following set of horn clauses with negation:

$$p(a), p(X) \wedge \neg q(X) \rightarrow r(X)$$

and written as disjunction:

$$p(a), \neg p(X) \vee q(X) \vee r(X).$$

This set of formulas has two minimal models:

$$M_1 = \{p\ (a), q(a)\} \text{ and } M_2 = \{p(a), r(a)\}.$$

A mechanism is needed to select one of these possible models. This model is then called the *perfect model* (Przymusinski 1988). The underlying assumption of a perfect model is as follows: Since there is no evidence for $p(X)$, we assume $\neg p(X)$ and therefore infer $r(X)$. Therefore, we choose $M_1 = \{p\ (a), r(a)\}$ as perfect model. Actually, we make use of the *syntactical structure of clauses* to select a minimal model as a perfect model.

More formal definitions can be found in Apt et al. (1988), Przymusinski (1988), Shepherdson (1988), Polleres (2006), Hitzler and Seda (2011), and Balke and Kroll (2020a).

14.1.4.4 Summary

In this section, we provided alternative semantics for FOL that improves the expressivity by using a different paradigm for interpretations and model theory. We introduced Herbrand semantics which provides a syntactical way to define interpretations and models derived from the language primitives used in a logical theory.

We can define (under certain restrictions) a model selector operator that provides a definite model from all possible models for Horn logic. In the simple case without negation, there is a *unique minimal model*. This extends the expressive power of first-order logic as we can now express the transitive closure of a relation. Facts that do not follow from our ground facts and axioms are assumed to be false. This sets the fundament for the so-called closed-world assumption.

For the cases where there are multiple minimal models, we introduced a mechanism that selects a *perfect model* based on a concept called stratification. There are still programs that contain recursion involving negated predicates, which are not stratifiable. However, there are also approaches to reason with non-stratifiable sets of formulas such as partial interpretations, unfounded sets, well-founded model semantics, and stable model semantics. However, these topics are beyond the scope of our book.[33]

14.1.5 Second-Order Logic and Further Logical Variants

There are various limitations to the expressivity of FOL under its standard semantics. In the previous section, we covered alternative semantics to address expressivity issues, particularly regarding transitive and deductive closure. These limitations and more may also be solved with higher-order logical languages. In this section, we will briefly introduce *second-order logic (SOL)*, particularly in the context of overcoming the limitation of not being able to make statements about statements, i.e., quantifying over not only elements of sets but also sets of sets (Keller 2004). We then introduce some other logical formalisms syntactically and/or semantically beyond first-order logic.

14.1.5.1 Second-Order Logic

Second-order logic (SOL) is a higher-order logic that extends FOL. FOL allows the quantification of elements in the universe. SOL allows quantification over sets of elements by allowing predicates to be applied to another predicate or formula. SOL extends the syntax of FOL with the following elements (Mekis 2016):

- Second-order predicates: They express properties and relations of properties and relations of individuals, e.g., P, Q, R, and S.
- Predicate variables: Their values are extensions of predicates, e.g., X, Y, and Z.

The formula production rule of FOL is extended with second-order predicates and predicate variables. Let R be a second-order predicate symbol and let X be a predicate variable. Let $T_1,\ldots, T_k$ be first-order predicate symbols or predicate variables and let $t_1,\ldots, t_n$ be first-order individual terms. Then,

$$R(T_1,, \ldots, T_k)$$

[33] See Eiter and Pichler (2010b) for some more details.

$$X(t_1, \ldots, t_n)$$

are atomic formulas.

Quantifiers are used as same as FOL except they are bound to the predicate variables and range over their extensions. For instance,

$$\forall X\, R(X, P) \rightarrow P(a)$$

is a second-order formula that can be read as "For all predicate X if predicate X is related to predicate P with relation R, then P is true for a." Here, X is a predicate variable, R is a second-order relation (binary predicate), P is a first-order unary predicate, and a is a constant (Mekis 2016).

FOL only allows quantification over individuals. SOL extends FOL with quantification over predicates. This leads to the ability to express things that cannot be expressed in FOL. If we want to express that two objects, a and b, have the same shape, there is not much we can do in FOL. One "hack" would be to enumerate all possible shapes as unary predicates and build a formula as a disjunction of conjunctions[34]:

$$(Cube(a) \wedge Cube(b)) \vee (Tet(a) \wedge Tet(b)) \vee (Dodec(a) \wedge Dodec(b))$$

With second-order logic, we can use a second-order predicate called *Shape* and quantify over a predicate P. The set of SOL formulas below express the objects a and b having the same shape "properly":

(1) $Shape(Cube) \wedge Shape(Tet) \wedge Shape(Dodec)$
(2) $\exists P\ (Shape(P) \wedge P(a) \wedge P(b))$

The first formula defines shapes called *Cube*, *Tet*, and *Dodec*. Here, *Shape* is a second-order predicate, while *Cube*, *Tet*, and *Dodec* are first-order predicate symbols. The second statement expresses that there is a shape, which is the shape of a and b at the same time.

More generally speaking, expressing there exists a predicate a and b have in common $\exists P\ (P(a) \wedge P(b))$ would require an infinite formula in FOL, which you also have to do when you "simulate" FOL with propositional logic.

SOL is also capable of representing the transitive closure of a relationship, as it can exclude unwanted models caused by standard FOL semantics. Details can be seen in Genereseth (2015) and Subramani (2017).

In the end, FOL fails to provide enough expressive power for formulating knowledge graphs, and SOL offers enough expressive power to formulate everything around knowledge graphs. As we have seen, making statements about statements and metamodelling can be crucial aspects of knowledge graphs, and FOL without certain "tricks" is not eligible for representing them. Unfortunately, nothing

[34] Example adapted from https://en.wikipedia.org/wiki/Second-order_logic

is free. FOL is the most expressive logical language that can be approached with computational means. SOL is beyond this boundary which means that under the standard semantics, there is no sound and complete proof system, which makes the logic computationally unapproachable. Fortunately, this is not the end of the road. There are certain logical formalisms that have higher-order syntactic features while still staying in the computational framework of FOL. For example, F-logic (Kifer and Lausen 1989) and HiLog (Chen et al. 1993), as we already discussed in Chap. 13, are such formalisms that provide syntactic features of SOL but have FOL semantics plus semantics of Herbrand models.

14.1.5.2 Other Logic Variants

What we have discussed so far is just a glimpse of the full picture. There are many other variants of logic with different characteristics. In this section, we briefly cover some of them.

Modal logic[35] is a logical formalism that allows talking about *necessity* and *possibility*. It introduces the $\Box$ and $\diamond$ operators for specifying the necessity and possibility of predicates, respectively. $\Box P \rightarrow \diamond P$ expresses that if P is necessary, then it is also possible. Modal logic can have different levels of complexity (PL, FOL, SOL).

Temporal logic[36] can be expressed by modal logic. It allows talking about the world in a temporal context. For example, the facts in a knowledge graph may not be true all the time, but can be true sometimes:

- **F***P*: It will sometimes be the case that *P* is true.
- **G***P*: It will always be the case that *P* is true.
- **P***P*: It was sometimes the case that *P* is true.
- **H***P*: It has always been the case that *P* is true.

Temporal logic can be point-based or interval-based. It has state- or history-based semantics.

Metalogic[37] usually has a logical language at the object layer and a meta-language to express statements about the language at the object layer. For example, one can use FOL at the object layer and an axiomatized proof system for it at the metalayer. You can use the metalayer to express and study different evaluation strategies applied to the object layer (see *Reasearch*).

An interesting observation is that RDF(S) allows arbitrary statements over statements; therefore, one would even need a *non-stratified Metalogic* to fully capture it (no surprise OWL-Full is not a computational logical language).

[35] https://en.wikipedia.org/wiki/Modal_logic

[36] https://en.wikipedia.org/wiki/Modal_logic#Temporal_logic

[37] https://en.wikipedia.org/wiki/Metalogic

14.1.6 Reasearch

Logical reasoning is considered one of the crucial parts of making use of the Semantic Web. However, there are severe contradictions between the nature of the Web and the assumptions of logic concerning completeness, correctness, and time (velocity) (Fensel and van Harmelen 2007). Logic is suitable for *small* sets of axioms and facts, but the knowledge on the Web is *very, very* large. Scaling reasoning for trillions of facts and axioms is beyond any scale of current logic reasoning engines. Taking the complete Web into account is non-feasible. Therefore, it is necessary to scope the reasoning on limited sub-fragments of the Web. *We need to rank information based on its relevance to our reasoning tasks.*

Traditional logic takes axioms as reflecting the truth and tries to infer implicit knowledge from them. Already one *inconsistency* allows us to infer any possible truth from any other fact. We all know the information on the Web is inherently inconsistent and often incorrect. Also, think about different contexts and points of view.

The Web is a *dynamic* entity with extremely high velocity. Therefore, we would need to freeze the Web during our reasoning time. Instead, by the time we finish an inference procedure, it is likely that our underlying rules and facts have already changed. The completeness requirement in a large and dynamic system does not make any sense. This is also generally known in system theory as completeness versus soundness (correctness) trade-off (Fig. 14.12).

"Reasearch" is an approach that fuses "reasoning" and "search." It replaces the notion of completeness and correctness with the notion of usability. Do I really need all the facts and axioms on the Web to obtain a usable inference? It considers reasoning as a process that takes the necessary resources into account. The main

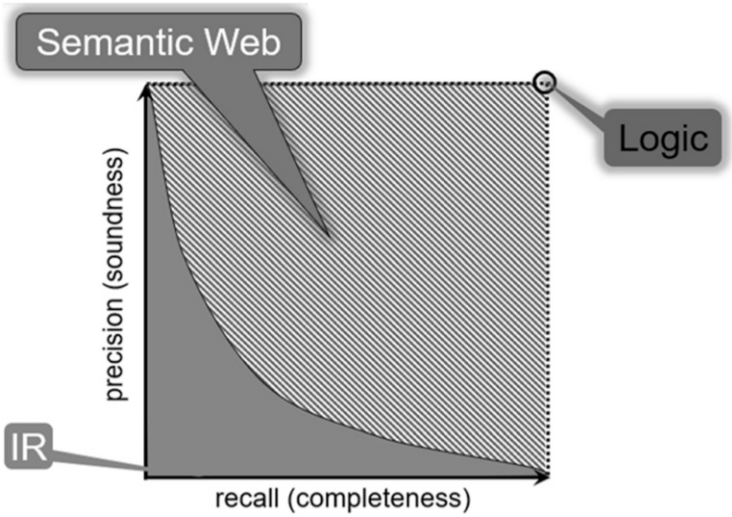


Fig. 14.12 The recall/precision trade-off of reasoning with limited rationality

```
do
        draw a sample,
        do the reasoning on the sample;
        if      you have more time, and/or if you don't trust the result,
        then draw a bigger sample,
repeat
```

Fig. 14.13 The reasearch algorithm

idea is to draw a number of triples and reason with them. Figure 14.13 shows the main algorithm of *reasearch.*

We can be smarter while selecting the sample to improve the usability of the inferences:

- Based on known distribution properties of the triples
- Based on their relation to the query
- Based on provenance properties like reputation or trust
- Based on experiences with previous queries

The approach is inspired by the idea of *limited rationality* from economy theory (Simon 1957; Newell and Simon 1972). It considers the cost of reaching the global optimum and finding a local optimum that works for your use case, just like we do not date with approximately eight billion people before we get married.

14.1.7 Summary

Logic can be used to formalize knowledge representation including knowledge graphs. It allows deriving true statements from statements that are considered to be true. For example, considering knowledge graphs as a set of such statements, what additional knowledge can we derive?

Propositional logic is the simplest logic. It has limited expressivity but also a decidable deduction process. FOL extends propositional logic with quantifiers, predicates, and function symbols. Semantics is defined based on the interpretation of the symbols over a domain of discourse (universe). The entailment procedures are semi-decidable: if there is a model, the entailment procedure can find it. If there is none, it can run forever. It is still the most reasonable logic in terms of applications in computer science. It is strongest in terms of expressivity by having enumerable semantics through its semi-decidability. Therefore, it has a complete and sound proof system.

There are subsets of FOL such as description logic that are decidable. Description logic is also the underlying formalism of the Web Ontology Language (OWL).

There are also alternative semantics such as Herbrand semantics that extend the expressivity of FOL. Herbrand models are a subset of all ground atoms. This is important for computational and conceptual reasons. With finitely many constants and without functions, FOL under Herbrand semantics is decidable. The intersection

of all Herbrand models is the minimal Herbrand model. This model is unique without negation: important for query-answering. In the presence of negation, mechanisms like perfect models must be used. Minimal model semantics is used in logic programming and deductive databases (e.g., Datalog and F-logic).

Higher-order logical languages extend FOL and eliminate many of its limitations. Second-order logic can make assertions about predicates. It allows the definition of transitive closure. There are also variants of logic that allow the representation of temporal facts and the provenance of facts. Higher-order logical languages are in general computationally not tractable. However, there are certain restricted (syntactic or semantic) higher-order logical languages that are decidable, e.g., monadic second-order logic for some theories and syntactic extensions of FOL, e.g., F-logic and HiLog.

14.2 RDF(S) Semantics

So far, we have already seen various logical formalisms (e.g., propositional logic, first-order logic, and its variations) that comprise the fundamentals of formal semantics. We know that formal semantics mainly deals with model theory; given a set of logical formulas, how can we infer new statements?

RDF(S) also has formal semantics that allows us to make inferences given some RDF data and schema. In this section, we will introduce the model theory for RDF (S) in terms of how its modelling primitives are interpreted and how new statements can be inferred from the existing ones with the help of entailment[38] rules.

In mathematics, "a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense related."[39] More formally, the mentioned set of objects is a set of vertices V, and the set of edges E contains pairs of vertices that are connected in the graph. Therefore, a graph G in the most generic sense can be defined as a pair $G = (V, E)$. RDF has a triple-based data model, as we have seen in Chap. 13. It can also be seen as a directed (multi)graph,[40] where the *subject* and *object* are vertices and each *predicate* is an ordered pair of a *subject* and an *object*.

RDF(S) has a rather unorthodox model theory unlike classical logical formalisms, as it first started as a data exchange format for the Web and then got formally defined. Therefore, the formal semantics of RDF graphs and RDFS language contains not only the interpretation of logical formulas but also graphs and XML datatypes. Therefore, as a convenience for understanding and implementation, the semantics is layered based on distinct modelling primitives. In the remainder of the section, we first show how interpretations and entailments are layered. Then, we explain the layers of

[38] If a set of formulas A entails another set of formulas B, then we say B is a logical consequence of A.

[39] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

[40] The same subject and object can be connected via multiple predicates; therefore, multigraph would be a more precise definition.

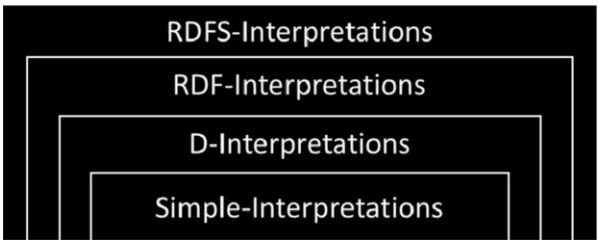


Fig. 14.14 The layered interpretations (and entailments) of RDF and RDFS

interpretations, followed by the layers of entailments. We will focus on the RDFS layer, as it is the most used one in practical applications. Finally, we provide a summary.

This section is based on the RDF 1.1 Semantics[41] specification and Hitzler et al. (2009). It provides only a gist of the RDF(S) semantics. We refer readers to the specification for more details. Additional details about the model theory of RDF (s) can also be found in Hitzler et al. (2009). Throughout this section, we make use of the syntax of RDF and RDFS. The readers may take back a quick look at Chap. 13 for a refresher on RDF(S).

14.2.1 RDF(S) Layering

Four layers structure RDF(S) interpretations and entailments.

- *Simple* interpretations and entailment provide the bottom layer. This layer provides the interpretations of the triple and graph model of RDF (<*subject*> <*predicate*> <*object*>).
- *Datatype* or D-interpretation and D-entailment builds on top of the simple layer and extends it with the interpretations and entailment based on the XML datatypes RDF is using (e.g., *xsd:string*).
- *RDF* interpretations and entailment build on top of the datatype layer with the interpretations and entailment based on the modelling primitives provided by the RDF vocabulary (e.g., *rdf:type*).
- *RDFS* interpretations and entailment is the final layer that extends the RDF layer with interpretations and entailment based on the RDFS language primitives (e.g., *rdfs:subClassOf*, *rdfs:domain*, *rdfs:range*).

Each layer will be explained in detail in the following sections (Fig. 14.14).

[41] This section contains content copied or derived from RDF 1.1 Semantics specification https://www.w3.org/TR/rdf11-mt Copyright © 2004-2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark, and document use rules apply.

14.2.2 RDF(S) Interpretations

In this section, we will explain how the primitives in RDF(S) are interpreted at the simple, datatype, RDF, and RDFS layers, respectively. Before we dive into the details, first, let us introduce some basic notions about interpretations and particularly how they give meaning to the symbols of an RDF graph.

An interpretation *I* satisfies a graph *G* if all symbols (IRIs, literals, blank nodes) in *G* are mapped to elements in the universe and all triples in the graph are mapped to pairs of resources that the property of the triple applies to (the property's extension). If *I* interprets a graph *G* as true, we write $I(G) = true$ or *I* satisfies *G* or *I* is a model of *G*. If there is no such interpretation that satisfies *G*, we say *G* is unsatisfiable.

We will use this terminology about interpretations in the following sections about different interpretation layers. Finally, we discuss what role containers play in the RDF(S) semantics.

14.2.2.1 Simple Interpretation

Simple interpretation deals with the interpretation of triples and the graph model of RDF. An RDF graph *V* is a set of triples. A triple is defined as $<s, p, o>$, where *s* is an IRI or a blank node; *p* is the predicate IRI that denotes a property, i.e., a binary relationship; and *o* is an IRI, a blank node, or a literal. A simple interpretation *I* of a graph *V* is defined by a 5-tuple:

$$I = < I_R, I_P, I_{EXT}, I_S, I_L >$$

- I_R is a finite, non-empty set of resources $\{d_1, d_2, \ldots, d_n\}$ called the domain or universe of *I*.
- I_P is a set that is called the set of properties of *I*. Note that I_R and I_P might overlap. For example[42]:

$$:\mathrm{x}\,:\mathrm{r_i}\,:\mathrm{y}.$$

$$:\mathrm{x}\,:\mathrm{z}\,:\mathrm{r_i}.$$

In this case, $I(:r_i)$ is both a member of I_R and I_P. I_P might even be a subset of I_R. This is the case when every relationship that is used in the predicate position is also used in the subject/object position of some triple.

- I_{EXT} is a *mapping* from I_P into $2^{I_R \times I_R}$, i.e., the set of sets of pairs $<x, y>$ with *x* and *y* in I_R. IEXT defines the extensions of binary relationships. For example,

[42] Throughout the section, *:* represents an empty prefix for a namespace.

given a property IRI p, the set of pairs of resources in I_R that are connected via $I(p)$ is in the extension of p.

- I_S is a *partial* mapping from IRIs in V[43] into $I_R \cup I_P$. A mapping from IRI references into the union of I_R and I_P., i.e., an IRI is interpreted as a resource in the universe or a property (or both, since the sets may overlap).
- I_L is a *partial* mapping from literals in V[44] into I_R. A literal name is interpreted as a resource (literal value to be precise) in the universe; however, the mapping is partial because some literals may not have a semantic value in the universe (see also D-interpretation) and also not all resources are literals.

So far, we only defined the interpretation of a graph with the consideration of IRIs and literals. However, RDF also provides *blank nodes*, which indicate the existence of a "thing" without using an IRI to identify it. Blank nodes correspond to existential quantification in logic ($\exists x$). To cover the interpretations of blank nodes, simple interpretations are extended as $[I+A]$ of the interpretation I with A. A is a mapping from a set of blank nodes to the universe I_R of I, with

$[I+A](x) = I(x)$ when x is an IRI or literal
$[I+A](x) = A(x)$ when x is a blank node

Blank nodes add a new semantic condition to an interpretation: If E is an RDF graph, then $I(E) =$ true if $(I+A)(E) =$ true for some mapping A from the set of blank nodes in E to I_R; otherwise, $I(E) =$ false.

14.2.2.2 D-Interpretation

D-interpretations deal with the correct handling of datatypes. Datatypes are identified by IRIs. Before we dive into the interpretations and semantic conditions, we need to introduce lexical-to-value mapping (*L2V*) that maps a lexical space onto a value space for a datatype. A datatype $d \in D$,[45] identified via an IRI, consists of a *value space* $V(d)$ (non-empty set of values) and a *lexical space* $L(d)$ (non-empty set of Unicode strings). The difference between a lexical space and value space must be done for datatypes because a single value can have different representations The lexical-to-value mapping *L2V* for a datatype d is a mapping from the lexical space to the value space: $L(d) \rightarrow V(d)$ (see Fig. 14.13).

[43] In the RDF(S) specification, the interpretations are defined in a more relaxed manner; in principle, that would require all possible IRIs to be interpreted. In practice however, only the IRIs in a finite graph are considered. See Sect. 14.2.2.5 about the finiteness of interpretations.

[44] Similar to the IRIs, RDF(S) specification defines interpretation of literals in a way that all possible lexical representations should be interpreted, which would theoretically lead to an infinite number of interpretations. In practice however, only the literals in a finite graph are considered. See Sect. 14.2.2.5 for details.

[45] D is the set of recognized RDF datatypes: https://www.w3.org/TR/rdf11-concepts/#section-Datatypes.

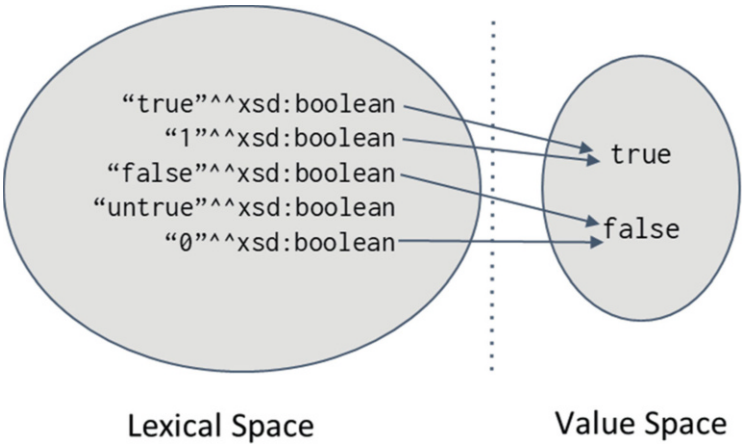


Fig. 14.15 The mapping between the lexical space and the value space of the *xsd:boolean* datatype

There are two important points to consider. First, RDFS 1.1 does not define the interpretation of untyped literals but assigns the *xsd:string* as datatype by default.[46] Second, there is a special datatype defined by RDF called *rdf:langString* whose interpretation is extended via a language tag. Literals with a language tag are of type *rdf:langString*. Other literals use a mapping from the so-called lexical-to-value mapping (L2V). Interpretation I D-satisfies graph G when the interpretation satisfies the semantic conditions of simple interpretations and the three new conditions:

- *Condition 1*: If *rdf:langString* in D, then for every language-tagged string s with lexical form *sss* and language tag *ttt*, $I_L(s) = \langle sss, ttt' \rangle$, where *ttt'* is *ttt* converted to lowercase according to US ASCII rules. A string with a language tag (e.g., @EN) maps to a corresponding pair in the universe with the language tag being translated to lowercase. For example, assume *rdf:langString* in D. Let s be a language-tagged string *"Austria@EN."* Then $I_L(s) = \langle Austria, @en \rangle$.
- *Condition 2*: For every other IRI, *aaa* in D, $I(aaa)$ is the datatype identified by *aaa*, and for every literal "*sss*"^^*aaa*, $I_L(\text{"}sss\text{"}\text{^^}aaa) = L2V(I(aaa), sss)$. Every literal with a certain datatype is mapped to a value for that datatype with the *L2V* mapping, given a lexical form of the literal. For example, let *aaa=xsd:boolean* and a literal *sss="0."*

$$I_L(\text{"0"}\text{^^}xsd{:}\ boolean) = L2V(I(xsd{:}\ boolean), \text{"0"})$$

$I_L(\text{"0"}\text{^^}xsd{:}\ boolean)$ maps to the value found under *"false"* in the *xsd:boolean* value space, according to the mapping shown in Fig. 14.15.

[46] https://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal

- *Condition 3*: An ill-typed literal "*sss*"^^*aaa* is a literal for which there is no mapping from the lexical space (*"sss"*) to the value space of the recognized type ("*aaa*"). If a triple of graph *G* contains an ill-typed literal, then *I(G) = false*. *G* is D-unsatisfiable.

14.2.2.3 RDF Interpretation

An interpretation *I* RDF-satisfies *G*, if it is a D-interpretation and it satisfies the two new semantic conditions as well as the set of axiomatic triples. The axiomatic triples are always interpreted as true for all RDF graphs (e.g., *rdf:type rdf:type rdf:Property*). We give a list of these axiomatic triples in Appendix 14.2.5.B-£ 14.2.5.2 with a brief explanation. In the following, we explain each semantic condition that an RDF interpretation must fulfill to satisfy a graph in an informal way. Formal definitions are given in Sect. 14.2.5.1.

- *Condition 1:* An IRI is interpreted as an RDF property if and only if that IRI is defined as an instance of *rdf:Property.*
- *Condition 2:* A literal value is an instance of a given datatype if and only if the value is in the value space of that datatype.

14.2.2.4 RDFS Interpretation

RDFS interpretation introduces new mappings that map RDFS modelling primitives to their meanings in the universe. An RDFS interpretation *I* makes an RDF graph true if it satisfies certain semantic conditions, in addition to the conditions provided by RDF interpretations as well as axiomatic RDFS triples. In the following, we will explain these conditions with the help of examples in more detail, since RDFS interpretations are the most used ones in practice. The formal definitions from the RDFS specification can be found in Appendix 14.2.5.3.

Condition 1: Every resource is an instance of *rdfs:Resource*. Consider a graph consisting of the following triple:

```
:harry :hasName "Harry Potter"^^xsd:string .
```

Then *:harry and "Harry Potter"* are both interpreted as instances of *rdfs:Resource.*

Condition 2: Every literal value is an instance of *rdfs:Literal*. Consider a graph consisting of the following triple:

```
:harry :hasName „Harry Potter"^^xsd:string.
```

The literal value in this triple "*Harry Potter*" is interpreted as an instance of *rdfs: Literal.* Note that *rdfs:Literal* is not the class of literals but literal values; for

example, although the string *"Harry Potter"* is in the extension of *rdfs:Literal*, the literal *"Harry Potter"^^xsd:string* is not.[47]

Condition 3: If y *is in the domain of x and* there is a triple in the RDF graph in the form of (*u, x, v*), then *u* is an instance of *y*. Consider a graph consisting of the following triples:

```
:hasParent rdfs:domain :Human .
:harry :hasParent :james .
```

The IRI *:harry* is interpreted as an instance of *:Human.*

Condition 4: If *y is in the range of x* and there is a triple in the RDF graph in the form of (*u x v*), then *v* has the type *y*. Consider a graph consisting of the following triples:

```
:hasParent rdfs:range :Human .
:harry :hasParent :james.
```

The IRI *:james* is interpreted as an instance of *:Human*

Condition 5: The *rdfs:subPropertyOf* property is reflexive and transitive over the set of properties in an RDF graph. This means that for every property *x*, *x* is a subproperty of itself. Also, if *rdfs:subPropertyOf* links property *x* with property *y* and also *y* with property *z*, then *rdfs:subPropertyOf* also links *x* directly with *z*. Consider the following property definitions:

```
:hasFather rdf:type rdf:Property.
:hasFather rdfs:subPropertyOf :hasParent .
:hasParent rdfs:subPropertyOf :hasAncestor.
```

The property *:hasFather* is interpreted as a subproperty of *:hasAncestor*, due to transitivity.

Condition 6: If two resources *x* and *y* are connected by *rdfs:subPropertyOf*, then both *x* and *y* are properties, and every pair of resources connected via *x* is also connected via *y*. Consider a graph consisting of the following triple:

```
:hasFather rdfs:subPropertyOf :hasParent.
```

[47] https://www.w3.org/TR/rdf11-mt/#a-note-on-rdfs-literal-informative

All subjects and objects connected via the *:hasFather* property in a triple are also connected by the *:hasParent* property.

Condition 7: Every class *x* is a subclass of the class of all resources, *rdfs:Resource*. Consider a graph consisting of the following triple:

```
:Human rdf:type rdfs:Class .
```

This implies that *:Human* is a subclass of *rdfs:Resource*.

Condition 8: If *x* and *y* are connected with the *rdfs:subClassOf* property, then both *x* and *y* are classes, and every instance of *x* is also an instance of *y*. Consider a graph consisting of the following triple:

```
:Man rdfs:subClassOf :Human .
```

This implies that *:Man* and *:Human* are classes and all instances of *:Man* are also instances of *:Human*.

Condition 9: The *rdfs:subClassOf* property is reflexive and transitive over the set of classes. If *x* is a class, then *x* is a subclass of itself. Also, if the *rdfs:subClassOf* property connects a class *x* with a class *y* and *y* with some class *z*, it also connects *x* with *z* directly. Consider a graph consisting of the following triples:

```
:Human rdf:type rdfs:Class .
:Man rdfs:subClassOf :Human .
:Human rdfs:subClassOf :Animal .
```

:Man is also a subclass of *:Animal*, due to the transitivity of the *rdfs:subClassOf* property.

Condition 10[48]: Any property that is an instance of *rdfs:Container-MembershipProperty* (e.g., *rdf:_1*, *rdf:_2*) is a subproperty of *rdfs:member* property. Consider a graph with the following triples:

```
:Friends_of_Harry_Potter rdf:type rdf:Seq .
:Ronald rdf:_1 :Friends_of_Harry_Potter.
:Hermione rdf:_2 :Friends_of_Harry_Potter.
```

[48] Although the containers are mostly about RDF, their semantic conditions are defined by RDFS interpretations.

:Ronald and *:Hermione* are also connected to *:Friends_of_Harry_Potter* with the *rdfs:member* property.

Condition 11: Any *x* typed as *rdfs:Datatype* is a subclass of the class of all literal values (denoted by *rdfs:Literal*). Consider a graph consisting of the following triple:

```
rdf:XMLLiteral rdf:type rdfs:Datatype .
```

rdf:XMLLiteral is a subclass of *rdfs:Literal.*

Like RDF, RDFS also has axiomatic triples that must be true in all RDFS interpretations, i.e., triples that are entailed by all graphs using the RDFS vocabulary (e.g., *rdfs:Resource rdf:type rdfs:Class*). A larger list of such triples can be found in Appendix 14.2.5.4.

14.2.2.5 Finiteness of RDF Interpretations and Containers

In its more generic sense, RDF semantics allows a larger, potentially infinite number of interpretations and models. For simple interpretations, all possible IRIs need to be interpreted, and for D-interpretations, already having *xsd:integer* or *xsd:string* must contain all integer numbers or all strings in the universe, which would lead to an infinite number of interpretations.

In practical implementations, however, RDF uses a stricter definition for these interpretations, that is, only the IRIs and literals that are used in an RDF graph are interpreted. This is also the reason why we defined the interpretations for IRIs (I_S) and literals (I_L) from a finite RDF graph *V* to IRIs and literals in the universe[49] for simple interpretations (Sect. 14.2.2.1) and D-interpretations (Sect. 14.2.2.2).

When talking about finiteness, one last thing to consider is the semantics of containers. RDF contains different types of containers. To recap from Sect. 13.1:

- *rdf:Bag* – a container for unordered items
- *rdf:Seq* – a container for ordered items
- *rdf:Alt* – a container for items that are alternative to each other

Formally, however, there is no semantics for the nature of these containers. For example, there is no formal definition of *rdf:Seq*, which defines its items to be ordered. The only formally defined aspect of containers is the membership property. RDF interpretations interpret membership property IRIs in the form of *rdf:_1*, *rdf:_2*... *rdf:_n*, which again does not formally indicate any ordering; however, it indicates that an element is a member of a container (see also Condition 10 in Sect. 14.2.2.4). Still, *n* being a positive integer, there can be infinitely many interpretations of this membership property. Also, in this situation, RDF

[49] See also the section in RDFS specification about finite interpretations: https://www.w3.org/TR/rdf11-mt/#finite-interpretations-informative.

interpretations only consider the finite number of membership properties that are used in the graph.[50]

14.2.3 RDF(S) Entailment

In this section, we explain the logical consequence relationship (*entailment*) between RDF graphs. This relationship is encoded with $G \vDash E$ where G and E are RDF graphs. Like the interpretations, the entailment of RDF graphs is also described in layers, each layer using the corresponding interpretation definitions for the entailment rules. We start with simple entailment and then follow with datatype, RDF, and, finally, RDFS entailment.

14.2.3.1 Simple Entailment

If a graph G simple entails a graph E, we say $G \vDash_{simple} E$. Every graph is *simple satisfiable*. This is because there is always a simple interpretation that makes the triples in an RDF graph true according to the definitions we made in Sect. 14.2.2.1. This does not hold for extended notions of interpretations, such as D-interpretations and other interpretations extending it. For instance, a graph containing an ill-typed literal is D-unsatisfiable. Simple entailment can be characterized by a single lemma called the *interpolation lemma*:

"G simple-entails a graph E if and only if a subgraph of G is an instance of E."

To understand this lemma, we need to first define the notions of *subgraph* and *graph instance*. A *subgraph* of an RDF graph is a subset of the triples in the graph. A triple is identified with the singleton set containing it so that each triple in a graph is a subgraph.

Any *graph E* obtained from a *graph G* by replacing some or all its blank nodes with RDF terms is *an instance of G*. Note that any graph is an instance of itself. An instance of an instance of G is an instance of G. If H is an instance of G, then every triple in H is an instance of some triple in G.

Note that the entailment (logical consequence) relationship is significantly different from how we explained it in the logic section. Simple entailment mixes the notions of model theory and graph theory; for instance, the core of entailment is defined based on the notion of "subgraph"; however, the definition of a subgraph does not treat each node equally due to the existence of blank nodes, which correspond to existentially quantified variables in logic.

[50] See also the section in the RDF(S) specification about containers: https://www.w3.org/TR/rdf11-mt/#rdf-containers.

There are several other lemmas that follow from the *interpolation lemma*. For example, the *empty graph lemma* states that "the empty graph is simple-entailed by any graph and does not simple-entail any graph except itself." Note that empty graphs are trivially true. If G is a ground graph,[51] then $I(G) = false$ if there is a triple T in G for which $I(T) = false$; otherwise, $I(G) = true$. Since empty graphs do not contain any triples, they also cannot contain any false triples.

The *subgraph lemma* says that "a graph simple-entails all its subgraphs." Similarly, the *instance lemma* states that "a graph is simple entailed by any of its instances."

There are two other important lemmas for simple entailment that follow from the interpolation lemma, namely, the *monotonicity* and *compactness lemmas.*

- The monotonicity lemma states that "if S is a subgraph of G, and S simple entails E, then G simple entails E." More informally, extending a graph does not make previous entailments invalid.
- On the other side of the coin, we have the compactness lemma, which states that "if G entails a finite graph E, then there is a finite subset S of G that entails E." In consequence, we say RDF is *compact*.[52] This feature may become important, considering that RDF graphs could be infinite.

14.2.3.2 D-Entailment

If a graph G D-entails a graph E, we say that $G \models_D E$.[53] D-entailment works with entailment rules about datatypes in literals. The entailment is based on the *L2V* mapping introduced in Sect. 14.2.2.2. Any triple containing a literal with a recognized datatype IRI (a member of the set D) D-entails another literal when the lexical strings of the literals map to the same by the lexical-to-value mapping of the datatype. As an example, we can revisit the lexical-to-value mapping in Fig. 14.15. Given that mapping, the entailments in Fig. 14.16 will follow.

There can also be cases where two different lexical strings with two different datatypes are mapped to the same value in the value space. This is, for example, typical if there is a certain hierarchy between datatypes. In this case, a graph containing a triple with one of these lexical strings may entail the other one. Figure 14.17 shows an example of this situation where a decimal literal entails an integer literal.

[51] A *ground graph* is a graph that only contains ground triples. A *ground triple* is a triple without blank nodes.

[52] Also, a very important property of first-order logic

[53] Implementation of reasoners to entail D-entailments can be implementation specific. Different reasoners may result in slightly different conclusions when D-entailment is checked.

G contains	G entails
ex:s ex:p “true”^^xsd:boolean .	ex:s ex:p “1”^^xsd:boolean .
ex:s ex:p “false”^^xsd:boolean .	ex:s ex:p “0”^^xsd:boolean .

Fig. 14.16 Example entailment given triples with literal value with xsd:boolean datatype

G contains	G entails
ex:s ex:p “42.0”^^xsd:decimal .	ex:s ex:p “42”^^xsd:integer .

Fig. 14.17 A datatype entails another datatype that maps different lexical strings to the same value

Fig. 14.18 Entailment rules for RDF entailment

name	G contains	G entails
rdf1	u a “s”^^d .	u a _:n . _:n rdf:type d .
rdf2	u a x .	a rdf:type rdf:Property .

14.2.3.3 RDF Entailment

If a graph *G* RDF-entails a graph *E*, we say that $G \models_{RDF} E$. RDF entailment works based on two entailment rules, shown in Fig. 14.18. These rules strongly follow the semantic conditions presented in Sect. 14.2.2.3. Note that axiomatic triples (Appendix 14.2.5.2) are always entailed.

The first rule *rdf1* entails the type of a literal (if the specified datatype *d* is an element of the set of datatypes *D*). We cannot directly infer *“s” rdf:type d*. This would not be a valid RDF triple since literals are not allowed in the subject position.

Therefore, we infer two triples: First is a triple stating that there is a value for the property *a* on resource *u*. Second, this value is the type of *d*.

The second rule *rdf2* states that the resource in the predicate position of a triple is an instance of *rdf:Property*.

14.2.3.4 RDFS Entailment

If a graph *G* RDFS-entails a graph *E*, we say that $G \models_{RDFS} E$. RDFS entailment works based on 13 entailment rules, shown in Fig. 14.19. Note that axiomatic triples (Appendix 4.2.5.C) are always entailed. In the following, we will first semi-formally introduce each rule and then provide a larger example that demonstrates how the RDFS entailment rules work. The entailment rules strongly follow the semantic conditions presented in Sect. 14.2.2.4.

The first rule *rdfs1* states that every datatype in *D* is of type *rdfs:Datatype*.

The rule *rdfs2* deals with the type inference based on the domain of a property. If *u* has *x* as a domain, and there exists a triple of the form "*y u z*," then *y* is of type *x*.

The rule *rdfs3* deals with the type inference based on a property range. If *u* has *x* as a range and there exists a triple of the form "*y u z*," *z* is of type *x*.

The rule *rdfs4* states that both the subject and object of a triple are an instance of *rdfs:Resource*.

The rule *rdfs5* deals with the transitivity of the subproperty relationship. If *u* is a subproperty of *x* and *x* is a subproperty of *z*, then *u* is also a subproperty of *z*.

The rule *rdfs6* deals with the reflexivity of the subproperty relationship. If *u* is of type *rdf:Property*, then *u* is a subproperty of itself.

The rule *rdfs7* deals with the inheritance of relationship instances (property extensions). Two resources, *y* and *z*, are related by property *x* if they are related by a subproperty of *x*. This corresponds to the bottom-up inheritance of instances (extensions) of properties.

The rule *rdfs8* states that every class is a subclass of *rdfs:Resource*.

The rule *rdfs9* deals with the bottom-up inheritance of class instances (class extensions). A resource *y* belongs to a class *x* if it belongs to its subclass *u*.

The rule *rdfs10* deals with the reflexivity of the subclass relationship. Every class is a subclass of itself.

The rule *rdfs11* deals with the transitivity of the subclass relationship. If *u* is a subclass of x and *x* is a subclass of *y*, then *u* is a subclass *y*.

The rule *rdfs12* is an entailment rule about container membership. A property *u* that is an instance of *rdfs:ContainerMembershipProperty* (e.g., *rdf:_1*, *rdf:_2*) is a subproperty of *rdfs:member*.

Finally, the rule *rdfs13* states that any resource *u* that is an instance of *rdfs:Datatype* is a subclass of *rdfs:Literal*.

Since it is the most used entailment layer in practice, we will demonstrate how the entailment rules of RDFS work with a concrete example. Figure 14.20 shows an example RDF graph *G*.

name	G contains	G entails
rdfs1	Any IRI d in D	d rdf:type rdfs:Datatype .
rdfs2	u rdfs:domain x . y u z .	y rdf:type x .
rdfs3	u rdfs:range x . y u z .	z rdf:type x .
rdfs4	u p x .	u rdf:type rdfs:Resource . x rdf:type rdfs:Resource .
rdfs5	u rdfs:subPropertyOf x . x rdfs:subPropertyOf z .	u rdfs:subPropertyOf z .
rdfs6	u rdf:type rdf:Property .	u rdfs:subProperty u .
rdfs7	u rdfs:subPropertyOf x . y u z .	y x z .
rdfs8	u rdf:type rdfs:Class .	u rdfs:subClassOf rdfs:Resource .
rdfs9	u rdfs:subClassOf x . y rdf:type u .	y rdf:type x .
rdfs10	u rdf:type rdfs:Class .	u rdfs:subClassOf u .
rdfs11	u rdfs:subClassOf x . x rdfs:subClassOf y.	u rdfs:subClassOf y .
rdfs12	u rdf:type rdfs:ContainerMembershipProperty .	u rdfs:subPropertyOf rdfs:member .
rdfs13	u rdf:type rdfs:Datatype .	u rdfs:subClassOf rdfs:Literal .

Fig. 14.19 RDFS entailment rules

```
:Human rdf:type rdfs:Class .
:Giant rdf:type rdfs:Class .
:hasParent rdf:type rdf:Property .
:hasParent rdfs:domain :Human .
:hasParent rdfs:domain :Giant .
:hasParent rdfs:range :Human .
:hasParent rdfs:range :Giant .

:hagrid :hasParent :fridwulfa .
```

Fig. 14.20 An example RDF graph *G*

Fig. 14.21 RDFS-entailed statements from G and by which rule each statement is entailed

#	G entails$_{RDFS}$	by rule
1	:Human rdfs:subClassOf rdfs:Resource .	rdfs8
2	:Giant rdfs:subClassOf rdfs:Resource .	rdfs8
3	:hagrid rdf:type rdfs:Resource .	rdfs4
4	:fridwulfa rdf:type rdfs:Resource .	rdfs4
5	:hagrid rdf:type :Human .	rdfs2
6	:hagrid rdf:type :Giant .	rdfs2
7	:fridwulfa rdf:type :Human .	rdfs3
8	:fridwulfa rdf:type :Giant .	rdfs3

Following the RDFS entailment rules, we can infer new statements based on the RDF graph *G*. Figure 14.21 shows some of the entailed triples and by which rule they are entailed. The first two triples are an application of *rdfs8*, which infers that both *:Human* and *:Giant* are subclasses of *rdfs:Resource*. The third and fourth inferred statements state that both *:hagrid* and *:fridwulfa* are instances of *rdfs:Resource*, based on the application of *rdfs4*. The fifth and sixth statements are inferred based on the rule *rdfs2*, because *:hagrid* is a subject of *:hasParent* and the property has *:Human* and *:Giant* classes in its domain. Similarly, the last two statements are inferred from the rule *rdfs3* as *:fridwulfa* is an object of *:hasParent* and the property has *:Human* and *:Giant* classes in its domain.

An interesting observation we can make with this example is that both *:hagrid* and *:fridwulfa* have two *rdf:type* assertions. RDFS treats domains and ranges as conjunctions of types they have. This means both :*hagrid* and *:fridwulfa* are instances of :*Giant* and *:Human* at the same time. Remember that RDFS does not provide any modelling primitive for defining class disjointness (unlike OWL); therefore, there is not much we can do about preventing such a situation except defining separate properties for the parents of giants and humans, with different ranges.[54]

14.2.4 Summary

An *interpretation* assigns meaning (semantics) to the terms (i.e., IRIs, literals, blank nodes) of a vocabulary and to sentences (i.e., RDF triples and graphs). A *model* of a set of expressions is an interpretation for which the set of expressions is evaluated to be true.

Interpretations in the context of RDF/RDFS consist of a set of resources, a set of properties, interpretation mappings, semantic conditions, and axiomatic triples. The meaning of a knowledge graph built with RDF(S) must be assigned via interpretations at different levels. *Simple interpretations* deal with the interpretation of triples. D-interpretations deal with the interpretation of datatypes. They allow an RDFS reasoner to ensure that literals are defined correctly with datatypes. RDF interpretations provide a very limited set of new conditions for providing meaning, mainly regarding the properties. RDFS interpretations provide the richest set of interpretation mappings that give meaning to class/property hierarchies and membership.

Interpretations are used to define the logical consequence (entailment) relationship. RDF(S) defines a layered set of entailment rules, like the interpretations.

- *Simple entailment* is defined in terms of graphs, subgraphs, and their instances. The entire simple entailment can be defined with the interpolation lemma: "*G* entails a graph *E* if and only if a subgraph of *G* is an instance of *E*." There are other semantic conditions that follow from this lemma.
- *D-entailment* enables the inference of literal values in triples based on the lexical-to-value space mappings defined by D-interpretations.
- *RDF entailment* provides entailment rules that infer predicates as properties and states the datatypes of literals.
- Finally, *RDFS entailment* provides 13 additional entailment rules for the inference of class/subclass relationships and their effects (class membership) on instances, property/subproperty relationships and effects on instances, domains/ranges of properties, type inference, and inferences based on the location of RDF terms in a triple.

[54] Considering the Harry Potter books, the intended meaning would match the inferences for Hagrid, but not for his mother Fridwulfa.

The RDFS entailment rules are widely used in practice; in fact, many knowledge graphs do not go beyond RDFS in terms of formalism.

14.2.5 Appendix

14.2.5.1 Appendix: RDF Semantic Conditions

- *Condition 1:* $x \in I_P$ if and only if $\langle x, I(rdf{:}\ Property)\rangle \in I_{EXT}(I(rdf{:}\ type))$.
- *Condition 2:* For every IRI aaa in D, $\langle x, I(aaa)\rangle \in I_{EXT}(I(rdf{:}\ type))$ if and only if $x \in V(I(aaa)\rangle$ where V(d) is the value space of the datatype d.

14.2.5.2 RDF Axiomatic Triples

The list below is taken from Section 8 of the RDFS Semantics 1.1 specification.[55] The axiomatic triples contain, among others, definitions of properties that are used for defining containers and their elements; for example, rdf:_1, rdf:_2, etc. are used to define members of a container, where *rdf:first* and *rdf:rest* are used to define list structures recursively.

```
rdf:type rdf:type rdf:Property .
rdf:subject rdf:type rdf:Property .
rdf:predicate rdf:type rdf:Property .
rdf:object rdf:type rdf:Property .
rdf:first rdf:type rdf:Property .
rdf:rest rdf:type rdf:Property .
rdf:value rdf:type rdf:Property .
rdf:nil rdf:type rdf:List .
rdf:_1 rdf:type rdf:Property .
rdf:_2 rdf:type rdf:Property .
...
```

14.2.5.3 Appendix: RDFS Semantic Conditions

Before we introduce the new semantic conditions that extend the RDF interpretation for RDFS, we introduce two new mappings as a shorthand, based on the definitions we made at simple and RDF interpretations:

[55] https://www.w3.org/TR/rdf11-mt/#rdf-interpretations

- We define $I_{instances}(y)$ as a shorthand for $\{x\text{: }(x,y) \text{ in } I_{EXT}(I(rdf\text{: } type))\}$. This means x is an instance of some y, if x is connected to y with the *rdf:type* property.
- Given this definition, we define $I_{classes}$ as $I_{instances}(I(rdfs\text{: } Class))$. This means $I_{classes}$ is the set of classes.

Condition 1: $I_{\text{instances}}(I(rdfs{:}Resource)) = I_{R.}$ Consider a graph consisting of the following triple*:*

```
:harry :hasName “Harry Potter”^^xsd:string .
```

The resources in this triple are interpreted as $I(:harry) \in I_{instances}(I(rdfs\text{: } Resource))$, I_R and I(“Harry Potter” ^ ^ xsd: string) $\in I_{instances}(I(rdfs\text{: } Resource), I_R)$.

Condition 2: $I_{instances}(I(rdfs{:}Literal)) = L_V$, where L_V is the set of literal values. Consider a graph consisting of the following triple:

```
:harry :hasName “Harry Potter”^^xsd:string .
```

The literal value in this triple is interpreted as I(“*Harry Potter*”) $\in$ $I_{instances}$ ($I(rdfs\text{: } Literal)$), L_V.

Condition 3: If $\langle x,\ y\rangle \in I_{EXT}(I(rdfs{:}domain))$ and $\langle u,\ v\rangle \in I_{EXT}(x)$, then $u \in I_{instances}(y)$. Consider a graph consisting of the following triples*:*

```
:hasParent rdfs:domain :Human .
:harry :hasParent :james .
```

The resource *:harry* here is then interpreted as $I(:harry) \in I_{instance}(I(:Human))$ because (I(:*hasParent*), I(:*Human*)) $\in I_{EXT}(I(rdfs\text{: } domain))$ and ($I(:harry)$, $I(:james)$) $\in I_{EXT}(I(:hasParent))$.

Condition 4: If $\langle x, y\rangle \in I_{EXT}(I(rdfs\text{: } range))$ and $\langle u, v\rangle \in I_{EXT}(x)$, then $v \in I_{instances}(y)$. Consider a graph consisting of the following triples:

```
:hasParent rdfs:range :Human .
:harry :hasParent :james .
```

The resource *:james* here is then interpreted as $I(:james) \in I_{instance}(I(:Human))$ because ($I(:hasParent), I(:Human)) \in I_{EXT}(I(rdfs\text{: } range))$ and ($I(:harry), I\ (:james)$) $\in I_{EXT}(I(:hasParent))$.

Condition 5: $I_{EXT}(I(rdfs{:}subPropertyOf))$ is transitive and reflexive on $I_{P.}$

```
:hasFather rdf:type rdf:Property .
:hasFather rdfs:subPropertyOf :hasParent .
:hasParent rdfs:subPropertyOf :hasAncestor .
```

This condition implies that $I_{EXT}(I(:hasFather)) \subseteq I_{EXT}(I(:hasFather))$, showing reflexivity, and $I_{EXT}(I(:hasFather)) \subseteq I_{EXT}(I(:hasParent))$ and $I_{EXT}(I(:Parent)) \subseteq I_{EXT}(I(:hasAncestor))$; therefore, $I_{EXT}(I(:hasFather)) \subseteq I_{EXT}(I(:hasAncestor))$ showing transitivity.

Condition 6: If $\langle x, y \rangle \in I_{EXT}(I(rdfs: subPropertyOf))$, then $x, y \in I_P$ and $I_{EXT}(x) \subseteq I_{EXT}(y)$. Consider a graph consisting of the following triple:

```
:hasFather rdfs:subPropertyOf :hasParent .
```

This condition implies that $I_{EXT}(I(:hasFather)) \subseteq I_{EXT}(I(:hasParent))$, which means all pairs of resources that are in the extension of property *:hasFather* are also in the extension of *:hasParent*.

Condition 7: If $x \in I_{\text{classes}}$, then $\langle x, I(rdfs:Resource) \rangle \in I_{EXT}(I(rdfs:subClassOf))$. Consider a graph consisting of the following triple:

```
:Human rdf:type rdfs:Class .
```

Given the definition of $I_{classes}$ at the beginning of the appendix, $(I(:Human), I(rdfs: Resource)) \in I_{EXT}(I(rdfs: subClassOf))$ because $I(:Human) \in I_{classes}$.

Condition 8: If $\langle x,y \rangle \in I_{EXT}(I(rdfs:subClassOf))$, then $x, y \in I_{\text{classes}}$ and $I_{\text{instances}}(x) \subseteq I_{\text{instances}}(y)$. Consider a graph consisting of the following triple:

```
:Man rdfs:subClassOf :Human .
```

This condition implies that $I_{instances}(I(:Man)) \subseteq I_{instances}(I(:Human))$, which means all resources that are in the class extension of *:Man* are also in the class extension of *:Human*.

Condition 9: $I_{EXT}(I(rdfs:subClassOf))$ is transitive and reflexive on I_{classes}. Consider a graph consisting of the following triples:

```
:Human rdf:type rdfs:Class .
:Man rdfs:subClassOf :Human .
:Human rdfs:subClassOf :Animal .
```

This condition implies that $I_{instances}(I(:Human)) \subseteq I_{instances}(I(:Human))$, showing reflexivity, and $I_{instances}(I(:Man)) \subseteq I_{instances}(I(:Human))$ and $I_{instances}(I(:Human)) \subseteq I_{instances}(I(:Animal))$; therefore, $I_{instances}(I(:Man)) \subseteq I_{instances}(I(:Animal))$ showing transitivity.

Condition 10: If $x \in I_{\text{instances}}(I(rdfs:ContainerMembershipProperty))$, then $\langle x, I(rdfs:member)\rangle \in I_{EXT}(\text{I}(rdfs:subPropertyOf))$.

This condition implies that $\langle I(rdf:_1), I(rdfs: member)\rangle \in I_{EXT}(I(rdfs: subPropertyOf))$ because $I(rdf:_1) \in I_{instances}(I(rdfs: ContainerMembershipProperty))$ (see the RDFS axiomatic triples in Appendix 14.2.5.4).

Condition 11: If $x \in I_{\text{instances}}(I(rdfs:Datatype))$, then $\langle x, I(rdfs:Literal)\rangle \in I_{EXT}$ $(I(rdfs:subClassOf))$. Consider a graph consisting of the following triple:

```
rdf:XMLLiteral rdf:type rdfs:Datatype .
```

This condition implies that $\langle I(rdf: XMLLiteral), I(rdfs: Literal)\rangle \in IEXT$ $(I(rdfs: subClassOf))$, because $I(rdf: XMLLiteral) \in Iinstances(I(rdfs: Datatype))$.

14.2.5.4 Appendix: RDFS Axiomatic Triples

The list below is taken from Section 9 of the RDFS Semantics 1.1 specification.[56] These triples extend the RDF axiomatic triples and contain, among others, definitions of some annotation properties provided by RDFS.

```
rdf:type rdfs:domain rdfs:Resource .
rdfs:domain rdfs:domain rdf:Property .
rdfs:range rdfs:domain rdf:Property .
rdfs:subPropertyOf rdfs:domain rdf:Property .
rdfs:subClassOf rdfs:domain rdfs:Class .
rdf:subject rdfs:domain rdf:Statement .
rdf:predicate rdfs:domain rdf:Statement .
rdf:object rdfs:domain rdf:Statement .
rdfs:member rdfs:domain rdfs:Resource .
rdf:first rdfs:domain rdf:List .
rdf:rest rdfs:domain rdf:List .
rdfs:seeAlso rdfs:domain rdfs:Resource .
rdfs:isDefinedBy rdfs:domain rdfs:Resource .
rdfs:comment rdfs:domain rdfs:Resource .
```

(continued)

[56] https://www.w3.org/TR/rdf11-mt/#rdfs-interpretations

```
rdfs:label rdfs:domain rdfs:Resource .
rdf:value rdfs:domain rdfs:Resource .

rdf:type rdfs:range rdfs:Class .
rdfs:domain rdfs:range rdfs:Class .
rdfs:range rdfs:range rdfs:Class .
rdfs:subPropertyOf rdfs:range rdf:Property .
rdfs:subClassOf rdfs:range rdfs:Class .
rdf:subject rdfs:range rdfs:Resource .
rdf:predicate rdfs:range rdfs:Resource .
rdf:object rdfs:range rdfs:Resource .
rdfs:member rdfs:range rdfs:Resource .
rdf:first rdfs:range rdfs:Resource .
rdf:rest rdfs:range rdf:List .
rdfs:seeAlso rdfs:range rdfs:Resource .
rdfs:isDefinedBy rdfs:range rdfs:Resource .
rdfs:comment rdfs:range rdfs:Literal .
rdfs:label rdfs:range rdfs:Literal .
rdf:value rdfs:range rdfs:Resource .

rdf:Alt rdfs:subClassOf rdfs:Container .
rdf:Bag rdfs:subClassOf rdfs:Container .
rdf:Seq rdfs:subClassOf rdfs:Container .
rdfs:ContainerMembershipProperty rdfs:subClassOf rdf:Property .

rdfs:isDefinedBy rdfs:subPropertyOf rdfs:seeAlso .

rdfs:Datatype rdfs:subClassOf rdfs:Class .

rdf:_1 rdf:type rdfs:ContainerMembershipProperty .
rdf:_1 rdfs:domain rdfs:Resource .
rdf:_1 rdfs:range rdfs:Resource .
rdf:_2 rdf:type rdfs:ContainerMembershipProperty .
rdf:_2 rdfs:domain rdfs:Resource .
rdf:_2 rdfs:range rdfs:Resource .
...
```

14.3 SPARQL Query Evaluation

So far, we have introduced the underlying logical formalisms for modelling languages for knowledge graphs. In this section, we will briefly introduce the semantics for SPARQL, which is based on the SPARQL algebra. We will first briefly explain what an algebra is. Then, we will introduce the core SPARQL algebra operations and their formal definitions. Afterward, we will show examples of how the abstract operations are evaluated on an RDF graph. Finally, we will conclude with a summary.[57]

14.3.1 Algebra

"In mathematics, an algebraic structure (or an algebra) consists of a non-empty set A (called the underlying set, carrier set or domain), a collection of operations on A of finite arity (typically binary operations), and *a finite set of equations, known as axioms*, that these operations must satisfy."[58] Following this definition, in the mathematical sense, we can define the following algebra: a non-empty set A, the binary operation (*), and closure axiom. Let us assume $A = \{-1,1\}$. The following operations can be defined within this algebra:

- $1 * 1 = 1$
- $1 * -1 = -1$
- $-1 * 1 = -1$
- $-1 * -1 = 1$

We can see that the closure requirements hold, as all the results of the multiplication operation are also an element of the set A. Contrarily, an algebra with the set A and summation operation (+) would not have the closure axiom since $1 + (-1) = 0$, and the result 0 is not an element of A.

As you may already have realized, the semantics of the operations defined on the sets are important for working with an algebraic structure. In the case of summation and multiplication, the semantics is relatively intuitive as they are arithmetic operators. If we would like to define an operator "#", on A, then we would also need to specify the meaning of this operator. That is, what should the operator do with the elements of the set? A common way to define such semantics is using set theory.

In computer science, a very well-known algebra is the *relational algebra*[59] defined by Codd (1970) for managing the data in relational databases. Relational algebra defines various operations on relations (tables), more precisely, the tuples

[57] The content of this section is largely based on SPARQL 1.1 specification: https://www.w3.org/TR/sparql11-query/#sparqlDefinition.

[58] Different examples for algebraic structures can be found here: https://en.wikipedia.org/wiki/Algebraic_structure.

[59] https://en.wikipedia.org/wiki/Relational_algebra

they contain. The most important feature of having an algebra is that the operations can be defined declaratively for "(multi)sets of tuples"; therefore, the meaning of the operations can be abstracted from implementations. For example, the Natural Join operation in relational algebra produces a Cartesian product of two sets (i.e., relations), and some elements are eliminated from the Cartesian product based on a join condition. We will present relational algebra in larger detail in Chap. 19.

Just like relational algebra defines operations on relations (tables), SPARQL algebra defines the meaning of SPARQL operations formally.[60] In the following section, we will explain the different operations of the SPARQL algebra.

14.3.2 SPARQL Algebra

A SPARQL query written in concrete syntax as we introduced in Sect. 13.2.1 must be converted into an abstract query before it is executed on an RDF graph. This helps SPARQL engines to create a canonical representation of the query that is independent from different syntactic variations. The transformation of a concrete query happens roughly in two steps:

1. Expansion of different syntactical forms (e.g., namespace prefixes to IRIs)
2. Converting clauses in concrete syntax to SPARQL algebra operations

In this section, we show a core subset of SPARQL algebra operations and their formal definitions. More precisely, we will cover the following operations that we informally introduced in Sect. 13.2.1:

- *BGP*: Corresponds to basic graph patterns in the concrete syntax.
- *Join:* There is no explicit concrete syntax correspondence of this operation, but it is used as an auxiliary to combine the results of other SPARQL algebra operations.
- *LeftJoin:* Corresponds to the OPTIONAL clause in the concrete syntax.
- *Union:* Corresponds to the UNION clause in the concrete syntax.
- *Filter:* Corresponds to the FILTER clause in the concrete syntax.

These operations are naturally just a small subset, but they represent the essence of SPARQL queries. For the remainder of the operations, like aggregation, ordering, and grouping of solutions, we refer readers to the SPARQL 1.1 specification. In the remainder of this section, we will first introduce the basic notions needed for defining the operations formally; then we will define each operation.

[60] https://www.w3.org/TR/sparql11-query/#sparqlDefinition

14.3.2.1 Basic Notions

In this section, we define the same essential notions that will be repeatedly used for explaining SPARQL algebra.

A *basic graph pattern* is a set of triple patterns. A triple pattern is a triple and member of the set defined as

$$\{(x, y, z) \mid x \in (RDFT \cup V), y \in (I \cup V), z \in (RDFT \cup V)\}$$

where *RDFT* is the set of RDF terms,[61] *V* is the set of query variables, and *I* is the set of IRIs. Also, the set of variables and RDF terms are disjoint:

$$(V \cap RDFT = \emptyset)$$

A *solution mapping* μ is a partial mapping from *V* to *RDFT*. $dom(\mu)$ is a set of variables $V' \subseteq V$, which corresponds to the domain of a solution mapping μ. *Two solution mappings,* μ_1 *and* μ_2*,are compatible*, if for every variable *v* in dom(μ_1) and in dom(μ_2), $\mu_1(v) = \mu_2(v)$.[62] Note that the definition of equality of RDF terms depends on the type of the RDF term. Two IRIs are equal if they are character-by-character the same Unicode string. Two literals are equal if and only if:

- Lexical forms of two strings compared character-by-character are equal
- Either both or neither has language tags and the language tags are equal strings if they exist
- Either both or neither has datatype IRIs and the datatype IRIs are equal strings if they exist

If μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ is also a solution mapping. Note that the solution mappings whose domains are disjoint ($dom(\mu_1) \cap dom(\mu_2) = \emptyset$) are always compatible (Pérez et al. 2009).

14.3.2.2 BGP

A basic graph pattern (*BGP*) operation produces a multiset of solution mappings from an RDF graph. Given a basic graph pattern *B*, and RDF graph *G*, the multiset of solution mappings produced by $BGP_G(B)$ is defined as Krötzsch (2019)

$$\Omega_{BGP_G(B)} = \{\mu \mid \mu(\sigma(B)) \subseteq G\}$$

[61] Reminder: An RDF term is an IRI, a blank node, or a literal. Although RDF does not allow literals in the subject position, SPARQL does not have such a restriction.

[62] https://www.w3.org/TR/sparql11-query/#func-RDFterm-equal

where $dom(\mu)$ is exactly the set of variables defined in B and σ is a mapping from blank nodes to RDF terms in G.[63] First the blank nodes in the basic graph pattern B are replaced with RDF terms in G, and then the solution mappings replace the variables with RDF terms in G. For each mapping of the variables to RDF terms, if the resulting triple exists in G, then the solution mapping is an element of $\Omega_{BGP_G(B)}$.

14.3.2.3 Join

Multiple BGPs and other algebraic operations that we will introduce later are handled with the help of the *Join* operation. A *Join* operation on two multisets of solution mappings Ω_1 and Ω_2, $Join(\Omega_1, \Omega_2)$, produces a new multiset of solution mappings $\Omega_{Join(\Omega_1,\Omega_2)}$ as follows:

$$\Omega_{Join(\Omega_1,\Omega_2)} = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \wedge \mu_2 \textit{ and } \mu_1 \textit{ are compatible}\}$$

In simpler words, the solution mappings in two multisets are merged with a set union if they have the same value for the variables they share.

14.3.2.4 LeftJoin

$LeftJoin_G$ is an operation on an RDF graph G that takes a multiset of solution mappings Ω_1 and adds solution mappings from another multiset Ω_2, if those solution mappings satisfy conditions of a *Join* operation and an optional Boolean filter expression *expr*.[64] Formally, the multiset of solutions produced by $LeftJoin_G(\Omega_1, \Omega_2, expr)$ is defined as

$$\Omega_{LeftJoin_G(\Omega_1,\Omega_2,expr)} = M_1 \cup M_2 \cup M_3$$

where:

$$M_1 = \{Join(\Omega_1, \Omega_2) \mid expr \textit{ evaluates true on } Join(\Omega_1, \Omega_2))$$

$$M_2 = \{\ \mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1 \textit{ and } \mu_2 \textit{ are not compatible} \vee \Omega_2 = \emptyset\}$$

$$M_3 = \{\ \mu_1 \mid \mu_1 \in \Omega_1, \exists \mu_2 \in \Omega_2, \mu_1 \textit{ and } \mu_2 \textit{ are compatible} \wedge expr \textit{ evaluates false on } \mu_1 \cup \mu_2\}$$

[63] We give the definition considering blank nodes for the sake of alignment with the specification; however, from now on, we assume the queries are blank node-free; therefore, the condition can be seen as $\mu(B) \subseteq G$.

[64] The reason the *LeftJoin* operation is defined over a graph G is the filter expression *expr*. There are a few filter expressions that need to access to the entire graph and not only the multisets of solution mappings at hand. For the sake of readability, the graph G is omitted in the conditions of the formal set definitions M_1 and M_3.

For a $LeftJoin_G(\Omega_1, \Omega_2, expr)$ operation, the order of the parameters matters, as the one on the "left" Ω_1 is extended with some elements of Ω_2.

14.3.2.5 Union

The *Union* operation on two multisets of solutions simply applies a set union on those multisets. The $Union(\Omega_1, \Omega_2)$ operation produces a multiset defined as follows:

$$\Omega_{Union(\Omega_1,\Omega_2)} = \{ \mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2 \}$$

14.3.2.6 Filter

The *Filter* operation uses a Boolean expression to filter out the solution mappings that do not satisfy that expression.

Given a Boolean filter expression *expr*, a multiset of solution mappings Ω, and an RDF graph G,[65] the $Filter_G(expr, \Omega, G)$ operation produces a multiset of solution mappings defined as follows:

$$Filter(expr, \Omega, G) = \{ \mu \mid \mu \in \Omega \text{ and } expr \text{ evaluates to true on } \mu \text{ (over } G) \}$$

14.3.3 Example Abstract Query Evaluation

In this section, we will explain how an abstract query that consists of SPARQL algebra operations is evaluated on an RDF graph, based on the definitions of operations we made in the previous section. Table 14.3 shows an RDF graph *G* in a tabular representation (the first column is not part of the data, but just an index for the row).

Figure 14.22 shows a query in concrete syntax that will be run over the RDF graph *G*.

Figure 14.23 shows an abstract representation of the same query. The two basic graph patterns (*?s schema:startDate ?sd* and *?s schema:endDate ?ed*) in Fig. 14.22 are converted into BGP operations (*BGP (?s schema:startDate ?sd)* and *BGP (?s schema:endDate ?ed)*), and the OPTIONAL clause is converted into a LeftJoin operation with a filter expression (*(year(?ed)= 2022)*). The filter expression checks whether the year of an end date is 2022.

[65] The RDF graph G is only necessary, if the Boolean filter expression *expr* needs to access to the entire RDF graph.

Table 14.3 The RDF graph on which the SPARQL query in Fig. 14.22 will be evaluated

Row#	Subject	Predicate	Object
1	dzt-entity:1501957001	schema: startDate	"2021-10-17T17:00:00.000Z"^^xsd: dateTime
2	dzt-entity:1501957001	schema: endDate	"2021-10-17T21:59:59.000Z"^^xsd: dateTime
3	dzt-entity:638252206	schema: startDate	"2022-03-26T09:00:00.000Z"^^xsd: dateTime
4	dzt-entity:638252206	schema: endDate	"2022-09-25T14:30:00.000Z"^^xsd: dateTime
5	dzt-entity:1857995432	schema: startDate	"2020-06-29T00:00:00+02:00"^^xsd: dateTime

```
PREFIX schema: <https://schema.org/>
SELECT * WHERE {

  ?s schema:startDate ?sd.
  OPTIONAL {
        ?s schema:endDate ?ed.
        FILTER (year(?ed)= 2022)
    }
```

Fig. 14.22 The example SPARQL query

The SPARQL query evaluation starts from the innermost operation and continues toward the outermost operation. Therefore, we will first present the evaluation of two *BGP* operations and then the *LeftJoin* operation that uses the results (multisets of solution mappings) of these two *BGP* operations to create the final multiset of solution mappings.

14.3.3.1 Evaluation of BGP Operations

Given the graph G in Table 14.3, the BGP_G(*?s schema*: *startDate* ? *sd*) maps variables *?s* and *?sd* to values in G, where all resulting triples are in G, after the replacement of these variables in ?*s schema*: *startDate* ? *sd* with values from G. This condition is only satisfied by the triples in rows 1, 3, and 5 in Table 14.3, as they have *schema:startDate* in the predicate position. The multiset of solution mappings after the operation is shown in Table 14.4.[66,67] Let us call this multiset Ω_1.

[66] Since we are currently working on multisets, the ordering is just for presentation purposes and done arbitrarily.

[67] Starting from Table 14.4, the first column of each table is an index for the sake of cross-referencing and not part of the multiset.

```
...

1   LeftJoin(
2    BGP (?s schema:startDate ?sd) ,
3    BGP (?s schema:endDate ?ed),
4   (year(?ed)= 2022)
5         )

...
```

Fig. 14.23 Abstract representation of the query in Fig. 14.22

Table 14.4 The multiset of solution mappings produced by BGP_G(?*s schema* : *startDate* ? *sd*)

μ_i	?s	?sd
μ_1	dzt-entity:1501957001	"2021-10-17T17:00:00.000Z"^^xsd:dateTime
μ_2	dzt-entity:638252206	"2022-03-26T09:00:00.000Z"^^xsd:dateTime
μ_3	dzt-entity:1857995432	"2020-06-29T00:00:00+02:00"^^xsd:dateTime

Table 14.5 The multiset of solution mappings produced by BGP_G(?*s schema* : *endDate* ? *ed*)

μ_i	?s	?ed
μ_4	dzt-entity:1501957001	"2021-10-17T21:59:59.000Z"^^xsd:dateTime
μ_5	dzt-entity:638252206	"2022-09-25T14:30:00.000Z"^^xsd:dateTime

Similarly, the operation BGP_G(?*s schema*: *endDate* ? *ed*) produces the multiset of solution mappings shown in Table 14.5. Let us call this multiset Ω_2.

14.3.3.2 Evaluation of LeftJoin

After we obtained Ω_1 and $\Omega_2$with the *BGP* operations, we evaluate the *LeftJoin* in the abstract query that takes these multisets and a filter expression as a parameter (*LeftJoin*(Ω_1, Ω_2, (*year*(?*ed*) = 2022)).

A *LeftJoin* operation produces a multiset of solution mappings that contains all solution mappings from the first multiset, in this case Ω_1, and extends it with some solution mappings from the second multiset, in this case Ω_2, based on some conditions.

Remember that a *LeftJoin* operation produces its resulting multiset in three steps:

1. Joins two multisets and applies the filter expression
2. Adds all non-compatible mappings from the first multiset that were left out after the first step
3. Adds all mappings from the first multiset that were eliminated after the first step by the filter expression

Table 14.6 The multiset of solution mappings after $Join(\Omega_1, \Omega_2)$

μ_i	?s	?sd	?ed
μ_6	dzt-entity:1501957001	"2021-10-17T17:00:00.000Z"^^xsd:dateTime	"2021-10-17T21:59:59.000Z"^^xsd:dateTime
μ_7	dzt-entity:638252206	"2022-03-26T09:00:00.000Z"^^xsd:dateTime	"2022-09-25T14:30:00.000Z"^^xsd:dateTime

In the following, we will demonstrate these three steps and reach our final result for the abstract query in Fig. 14.23.

Step 1: Join and Apply the Filter Expression
As described in Sect. 14.3.2.4, the formal definition of this step is

$$\{Join(\Omega_1, \Omega_2) \mid expr\ evaluates\ true\ on\ Join(\Omega_1, \Omega_2))$$

The *Join* operation takes two solution mappings from two multisets and merges them with a set union into a new solution mapping, if they are compatible (i.e., have the same value for the same variables). The resulting solution mappings are elements of a new multiset, which is the result of the *Join* operation.

Let us join the two multisets, Ω_1 and Ω_2, shown in Tables 14.4 and 14.5, respectively. The solution mappings in both multisets have one common variable, namely, *?s*. There are no compatible solution mappings in Table 14.5 for μ_3. The following solution mappings are compatible; therefore, they are merged with a set union into new solution mappings as members of the multiset shown in Table 14.6:

- μ_1 is compatible with μ_4 (over the value dzt-entity:1501957001).
  $\mu_6 = \mu_1 \cup \mu_4$
- μ_2 is compatible with μ_5 (over the value dzt-entity:638252206).
  $\mu_7 = \mu_2 \cup \mu_5$

Let us call the resulting multiset of $Join(\Omega_1, \Omega_2)$ operation Ω_3 (Table 14.6).

After the Join operation, the filter expression *((year(?ed)= 2022)* is applied to the solution mappings in Ω_3. The filter expression produces a new multiset of solution mappings containing the solution mappings that satisfy the expression. In this case, μ_6 evaluates the expression false, and only μ_7 in Table 14.6 satisfies the condition, which says the year component of the value of the variable *?ed* must be 2022. Table 14.7 shows the new multiset created after filtering. Let us call this multiset Ω_4.

Step 2: Adding Non-compatible Solution Mappings from the First Multiset
As described in Sect. 14.3.2.4, the formal definition of this step is

$$\{\ \mu_1 \mid \mu_1\ in\ \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1\ and\ \mu_2\ are\ not\ compatible \vee \Omega_2 = \emptyset\}$$

During Step 1, the non-compatible solution mapping μ_3 in Table 14.4 (Ω_1) was left out due to the definition of the *Join* operation. At this step, exactly that solution mapping is added to the multiset of solutions created after Step 1.

Table 14.7 The multiset of solution mappings after applying the filter expression on $Join(\Omega_1, \Omega_2)$

μ_i	?s	?sd	?ed
μ_7	dzt-entity:638252206	"2022-03-26T09:00:00.000Z"^^xsd:dateTime	"2022-09-25T14:30:00.000Z"^^xsd:dateTime

Table 14.8 The multiset of solution mappings after adding the non-compatible solution mapping from Ω_1

μ_i	?s	?sd	?ed
μ_7	dzt-entity:638252206	"2022-03-26T09:00:00.000Z"^^xsd:dateTime	"2022-09-25T14:30:00.000Z"^^xsd:dateTime
μ_3	dzt-entity:1857995432	"2020-06-29T00:00:00+02:00"^^xsd:dateTime	

The resulting multiset

$$\Omega_5 = \Omega_4 \cup \mu_3$$

is shown in Table 14.8.

Step 3: Adding the Solution Mappings That Were Eliminated from $Join(\Omega_1, \Omega_2)$ due to the Filter Expression from the First Multiset
As described in Sect. 14.3.2.4, the formal definition of this step is

$$\{\, \mu_1 \mid \mu_1 \in \Omega_1, \exists \mu_2 \in \Omega_2, \mu_1 \textit{ and } \mu_2 \textit{ are compatible} \wedge \textit{expr evaluates false on } \mu_1 \cup \mu_2 \}$$

In Step 1, μ_6 was built as $\mu_1 \cup \mu_4$, since two solution mappings were compatible. However, μ_6 was later eliminated because it evaluated the filter expression as false. According to the formal definition, the component of μ_6 that comes from Ω_1, namely, μ_1, is added to Ω_5. This leads us to the final result of our query

$$\Omega_6 = \Omega_5 \cup \mu_1$$

which is shown in Table 14.9.

Table 14.9 also supports the intuitive definition of the *LeftJoin* operation: We have all solution mappings from Ω_1 (the parameter on the left-hand side of the *LeftJoin* operation) and extend it with one solution mapping from Ω_2 that satisfies the conditions of the *Join* operation and the filter expression.

Table 14.9 The multiset of solution mappings Ω_6 as the final result

μ_i	?s	?sd	?ed
μ_7	dzt-entity:638252206	"2022-03-26T09:00:00.000Z"^^xsd:dateTime	"2022-09-25T14:30:00.000Z"^^xsd:dateTime
μ_3	dzt-entity:1857995432	"2020-06-29T00:00:00+02:00"^^xsd:dateTime	
μ_1	dzt-entity:1501957001	"2021-10-17T17:00:00.000Z"^^xsd:dateTime	

14.3.4 Summary

SPARQL is the query language for RDF standardized by the W3C and consequently applicable to knowledge graphs modelled with RDF. In this section, we briefly explained how SPARQL semantics is defined with the help of the SPARQL algebra. We first introduced algebra in general and explained the motivation for having algebra for a query language like SPARQL. Then we defined the formal meaning of various core SPARQL algebra operations. We refer readers to the SPARQL specification for the definitions of other SPARQL algebra operations.[68]

Then, we demonstrated how an actual abstract query is evaluated on an RDF graph based on the formal definitions of operations. Note that there are many query optimization techniques that may have an impact on how an abstract query is built and evaluated. However, our main goal with this section was not to present such approaches but to show the principles and core of the SPARQL algebra defining semantics for SPARQL query evaluation. Other more detailed presentations of the SPARQL algebra with examples can be found in Krötzsch (2019), Hitzler et al. (2009), and Sack (2015).

14.4 Summary

There are many languages for modelling and querying knowledge graphs that we have already introduced in Chap. 13. These languages allow transferring human knowledge to computers, which gives a mechanism that allows computers to understand the modelled knowledge. This mechanism is provided by formal semantics, where meaning is encoded by mathematical means such as logic or algebra. We covered formal semantics of the languages for knowledge graph reasoning, modelling, and querying in three sections.

First, we introduced *logic* as a basis for giving meaning to knowledge graphs. We first introduced the most basic logic, namely, propositional logic and fundamental notions such as interpretations, logical entailment and proof, as well as how

[68] https://www.w3.org/TR/sparql11-query/#sparqlDefinition

automated proofs can be created. Although decidable, propositional logic is very limited in terms of expressivity for knowledge representation. We introduced first-order logic that addresses the limitations of propositional logic while suffering at the decidability front. We then introduced two different camps for modifying first-order logic. These two modifications aim to find a sweet spot in terms of expressivity and decidability. One of them is description logic, a decidable binary subset of first-order logic, which is the underlying formalism of OWL. The second one provides an alternative definition of interpretation, models, and logical entailment and replaces the open-world assumption of FOL with a closed-world assumption. This modification is based on Herbrand models and comprises the foundation of rule languages such as Datalog and languages like F-logic.

Second, we explained the *formal semantics* of the *RDF(S)* language for modelling knowledge graphs. RDF(S) has an interesting approach to defining formal semantics that mixes the notions from graph theory and logic. The semantics is defined in a layered manner starting from simple interpretations and entailment which essentially defines the notion of entailment in terms of graph theory and ends with the RDFS interpretations and entailment that offers 13 rules that allow inferring new knowledge from existing knowledge graphs, mainly focusing inference of type and property assertions using class hierarchy and property domains and ranges.

Finally, we presented the formal semantics of the query language for RDF-based knowledge graphs, SPARQL. SPARQL queries are evaluated on an RDF graph based on certain operations that are offered by the *SPARQL algebra.* The meaning of these operations, i.e., how they manipulate an RDF graph, is formally defined as operations on multisets of solution mappings.

Perhaps it is evident that Chap. 14 is structured in parallel to Chap. 13 as we provide the underlying formalisms for the languages we presented in Chap. 13. Nevertheless, there are exceptions for SHACL and SKOS, whose formal underpinnings are not mentioned in this section. This situation is not a coincidence, as SHACL does not have official formal semantics and is typically expressed by SPARQL queries. The primary purpose of SKOS is not to enable reasoning but to provide a bridge between the formal world of OWL and the informal and chaotic world of the Web.

At the beginning of Part II, we mentioned that the *epistemological layer* provides a link between conceptual modelling (i.e., intended meaning by humans) and the *formal layer*. With this chapter on logic, we provided one end of this link. In the next chapter, we will cover a well-adopted ontology, namely, schema.org, which is very important for modelling knowledge graphs. Actually, here, we define the opposite link, i.e., toward the human user, which is provided by the *conceptual layer*.

References

Apt KR, Blair HA, Walker A (1988) Chapter 2 - Towards a theory of declarative knowledge. In: Minker J (ed) Foundations of deductive databases and logic programing. Morgan Kaufmann, pp 89–148. https://doi.org/10.1016/B978-0-934613-40-8.50006-3

Baader F (2009) Description logics. In: Tessaris S, Franconi E, Eiter T, Gutierrez C, Handschuh S, Rousset MC, Schmidt RA (eds) Reasoning Web. Semantic Technologies for Information Systems. 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30-September 4, 2009, Tutorial Lectures, Springer, pp 1–3

Baader F, Calvanese D, McGuinness DL, Nardi D, Patel-Schneider PF (eds) (2003) The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York

Bachmair L, Ganzinger H (2001) Resolution theorem proving. In: Handbook of automated reasoning. Elsevier, pp 19–99

Balke W, Kroll H (2020a) Lecture 5 - Knowledge-based systems and deductive databases lecture notes. http://www.ifis.cs.tu-bs.de/sites/default/files/KBS WiSe20 v5.pdf

Balke W, Kroll H (2020b) Lecture 7 - Knowledge-based systems and deductive databases lecture notes. http://www.ifis.cs.tu-bs.de/sites/default/files/KBS WiSe20 v7.pdf

Bancilhon F, Maier D, Sagiv Y, Ullman JD (1985) Magic sets and other strange ways to implement logic programs. In: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on principles of database systems, Cambridge, MA, USA, March 24–26, pp 1–15

Ben-Ari M (2012) Mathematical logic for computer science. Springer

Brachman RJ, Levesque HJ (2004) Chapter 4: Resolution. In: Representation and reasoning. Elsevier/Morgan Kaufmann, San Francisco, CA

Bruni C (2018) Propositional logic: resolution. cs245 lecture slides. https://cs.uwaterloo.ca/~cbruni/CS245Resources/lectures/2018_Fall/06_Propositional_Logic_Resolution_Part_1_post.pdf

Codd EF (1970) A relational model of data for large shared data banks. Commun ACM 13(6): 377–387

Chen W, Kifer M, Warren DS (1993) Hilog: a foundation for higher-order logic programming. J Logic Program 15(3):187–230

Dyer CR (1998) First-order logic. CS540 lecture notes. http://pages.cs.wisc.edu/dyer/cs540/notes/fopc.html

Eiter T, Pichler R (2010a) Foundations of rule and query languages. lecture slides of foundations of data and knowledge systems VU 181.212. https://www.dbai.tuwien.ac.at/education/fdks/dks04-2x2.pdf

Eiter T, Pichler R (2010b) Declarative semantics of rule languages. Lecture slides of foundations of data and knowledge systems VU 181:212. https://www.dbai.tuwien.ac.at/education/fdks/dks05-2x2.pdf

Fensel D, Van Harmelen F (2007) Unifying reasoning and search to web scale. IEEE Internet Comput 11(2)

Fitting M (1996) First-order logic and automated theorem proving. Springer-Verlag, New York

Genereseth M (2015) Herbrand manifesto: Thinking inside the box. In: Keynote talk in the 9th International Web Rule Symposium, Berlin, Germany, August 3–5. https://conference.imp.fu-berlin.de/cade-25/download/2015 CADEruleml genesere.pdf

Hitzler P, Seda A (2011) Mathematical aspects of logic programming semantics. Taylor & Francis

Hitzler P, Krotzsch M, Rudolph S (2009) Foundations of semantic web technologies. CRC Press

Horrocks I (2005) Description Logic reasoning at the International Conference on Logic Programming and Automated Reasoning (LPAR), Montevideo, Uruguay. http://www.cs.ox.ac.uk/ian.horrocks/Seminars/download/semtech-tutorial-pt3.pdf

Horrocks I, Sattler U (2007) A tableau decision procedure for. J Autom Reason 39(3):249–276

Huth M, Ryan M (2004) Logic in computer science, 2nd edn. Cambridge University Press

Keller U (2004) Some remarks on the definability of transitive closure in first-order logic and datalog. Technical report, Digital Enterprise Research Institute (DERI), University of Innsbruck

Kifer M, Lausen G (1989) F-logic: A higher-order language for reasoning about objects, inheritance and schema. In: SIGMOD/PODS04: International Conference on Management of Data and Symposium on Principles Database and Systems, Portland, Oregon, USA, June 1, pp 134–146

Krötzsch M (2019) Lecture 6: SPARQL semantics. Lecture slides of knowledge-based systems. https://iccl.inf.tu-dresden.de/w/imagses/3/3e/KG2019-Lecture-06-overlay.pdf

Mekis P (2016) Second-order logic. Technical report. http://lps.elte.hu/mekis/sol.pdf

Newell A, Simon HA (1972) Human problem solving, vol 104. Prentice-Hall, Englewood Cliffs, NJ

Pérez J, Arenas M, Gutierrez C (2009) Semantics and complexity of SPARQL. ACM Trans Database Syst 34(3):1–45

Polleres A (2006) Lecture 9: Alternative semantics for negation: perfect, well-founded and stable models. https://aic.ai.wu.ac.at/polleres/teaching/lma2006/lecture9.pdf

Przymusinski TC (1988) Chapter 5 - On the declarative semantics of deductive databases and logic programs. In: Minker J (ed) Foundations of deductive databases and logic programming. Morgan Kaufmann, pp 193–216. https://doi.org/10.1016/B978-0-934613-40-8.50009-9

Rudolph S (2011) Foundations of description logics. In: Reasoning web semantic technologies for the web of data: 7th International Summer School 2011, Galway, Ireland, August 23–27, 2011, Tutorial Lectures 7, pp 76–136

Sack H (2015) 2.10 Extra: SPARQL data management and algebra. OpenHPI tutorial - knowledge engineering with semantic web technologies. https://www.youtube.com/watch?v=W2aEb7mbi0Q

Shepherdson JC (1988) Chapter 1 - Negation in logic programming. In: Minker J (ed) Foundations of deductive databases and logic programming. Morgan Kaufmann, pp 19–88. https://doi.org/10.1016/B978-0-934613-40-8.50005-1

Schöning U (2008) Logic for computer scientists. Springer

Simon HA (1957) Models of man: social and rational-mathematical essays on rational human behavior in a social Setting. Wiley

Subramani K (2017) Introduction to second-order logic. Computational complexity lecture notes. https://community.wvu.edu/~krsubramanicourses/sp09/cc/lecnotes/sol.pdf

Chapter 15
Analysis of Schema.org at Five Levels of KR



In the previous chapters, we introduced five levels of knowledge representation, namely, implementation, logical, epistemological, conceptual, and linguistic levels. So far, we have focused on the epistemological and logical levels. Ontologies are used to provide reusable schemas for the conceptual level. We introduced ontologies in Part I and will cover them again more extensively in Sect. 18.2, where we talk about knowledge creation. In this chapter, we comprehensively analyze a de facto industrial standard ontology called *schema.org* in the framework of these five levels.[1] This chapter will demonstrate the use of these levels in practice.

We will start the analysis of schema.org at the conceptual level as an ontology. Then we will introduce the epistemological decisions made while designing schema.org. Then we will introduce two different ways of representing the ontology at the logical level. Finally, we will introduce several syntaxes to encode schema.org at the implementation level.

15.1 Schema.org at the Conceptual Level

Schema.org has been initiated by four big search engines, Bing, Google, Yahoo!, and Yandex, to allow content publishers on the Web to annotate their content and data semantically so the search engines can understand them better.

Schema.org contains[2] 811 types, in two disjoint sets. Seven hundred ninety-seven types are more specific than Thing, and 14 types are more specific than Datatype. It has 1453 properties and 86 enumerations with 462 enumeration members. Schema.org is the de facto standard for describing things on the Web and actions that can be taken on them. Figure 15.1 shows the direct subtypes of *schema:Thing* and schema:

[1] See also Patel-Schneider (2014) for a similar analysis.

[2] https://schema.org/docs/schemas.html. Last accessed: October 2022.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_15



Fig. 15.1 Direct subtypes of *schema:Thing* and *schema:DataType*

DataType types. For example, the *Event* type describes "an event happening at a certain time and location, such as a concert, lecture or festival," and can have properties describing its *start date*, *end date*, *location*, and *organizer*. Schema.org contains types and properties that cover a variety of domains that have commercial value for the consortium behind schema.org.

15.2 Schema.org at the Epistemological Level

Epistemological primitives and patterns used by schema.org are types (generic concepts), properties (attributes, roles), inheritance relationships, instantiation relationships, and multityped entities. In the following, we will explain how these epistemological primitives are being used by schema.org.

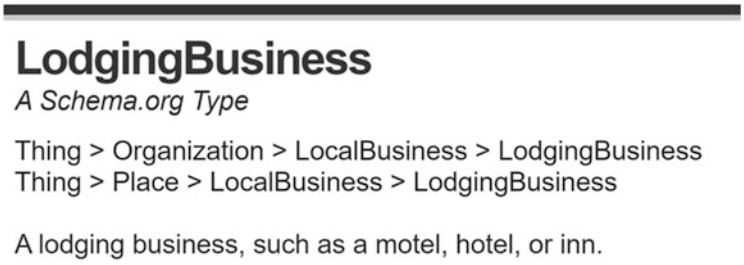


Fig. 15.2 *schema.LodgingBusiness* type has two distinct immediate super types

15.2.1 Types

Ontologies typically have classes to define a set of instances. In the Schema.org terminology, classes are called types. Schema.org organizes types into three disjoint hierarchies:

- First, there are types that are more specific than *schema:Thing*, for example, *schema:Action*, *schema:CreativeWork*, *schema:Event*, *schema:Intangible*, *schema:MedicalEntity*, *schema:Organization*, *schema:Person*, *schema:Place*, *schema:Product*, and their subtypes.
- There are types that are more specific than *schema:DataType*, for example, *schema:Boolean*, *schema:Date*, *schema:DateTime*, *schema:Number*, *schema:Text*, and *schema:Time*.
- Finally, there is a special hierarchy of types more specific than *schema:Enumeration*, for example, *schema:BookFormatType*, *schema:CarUsageType*, *schema:DayOfWeek*, and *schema:DeliveryMethod*. For example, the *schema:DayOfWeek* enumeration type has a predefined set of members consisting of:
  - *schema:Monday*
  - *schema:Tuesday*
  - *schema:Wednesday*
  - *schema:Thursday*
  - *schema:Friday*
  - *schema:Saturday*
  - *schema:Sunday*
  - *schema:PublicHolidays*

Each type can have more than one immediate supertype, which means the hierarchy is not a tree. For instance, *schema:LocalBusiness* is a subtype of both *schema:Organization* and *schema:Place* types (Fig. 15.2).

15.2.2 Properties

Properties are primitives that link two types with domain and range definitions. Like types, they are organized in a hierarchy in schema.org. The domain of a property defines the types on which a property is used. The range defines the expected types for property values. Two instances that are connected via a property comprise an instantiation of the property.

Schema.org properties can be interpreted from two different perspectives, namely, the property-oriented perspective and the frame-based perspective. From *the property-oriented perspective (global properties)*, properties are globally defined. Each type in a domain is valid for each type in the range, which creates a Cartesian product of domains and ranges (due to the disjunctive nature of domains and ranges, which we will discuss below). For example, see the *schema:address* property shown in Fig. 15.3. The following statements about the domain and range definitions are correct:

- The *schema:address* property has type *schema:Place* as domain and type *schema:PostalAddress* as range.
- The *schema:address* property has type *schema:Place* as domain and type *schema:Text* as range.
- The *schema:address* property has type *schema:Person* as domain and type *schema:PostalAddress* as range.
- The *schema:address* property has type *schema:Person* as domain and type *schema:Text* as range.
- . . .

From *the frame-based perspective (local properties)*, properties are defined locally on types. Each property can have a specific range for a given domain (there can still be multiple types in the range):

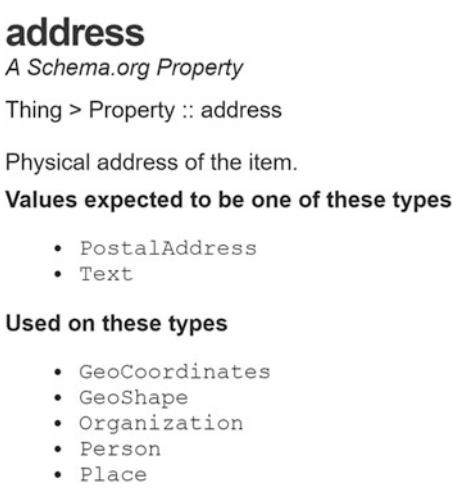


Fig. 15.3 The *schema:address* property seen from the property-oriented perspective

- The type *schema:Place* has the *schema:address* property that can take *schema: PostalAddress* instances as value.
- The type *schema:Person* has the *schema:address* property that can take *schema: Text* or *schema:PostalAddress* instances as value.

The most important thing to note about the properties in schema.org is that domains and ranges are defined by the disjunctions certain types.

The conjunction of two types implies a set intersection between the sets represented by these two types. Given two types, A and B (as domain or range definition), if i is an instance of the conjunction of A and B, then $i \in A \cap B$.

The disjunction of two types (as domain or range definition), however, implies that if i is an instance of the disjunction of A and B, then $i \in A \cup B$.

The disjunctive definition of domains and ranges has a significant impact on other epistemological primitives, like the inheritance relationship and its implication on the logical layer, which we will cover in Sect. 15.3.

15.2.3 Inheritance

Inheritance relationships between types have two major consequences. First, inheritance enables a type to inherit all properties of its supertypes. This also implies the inheritance of range definitions of these properties. Subtypes can also further restrict property ranges (in the frame-based interpretation). Second, inheritance enables types to inherit all instances from their subtypes. Every instance of a subtype is also an instance of a supertype.

In schema.org, ranges are defined disjunctively, i.e., a subclass can extend the ranges of inherited properties in the frame-based setting. The implications of this situation at the logical level will be discussed in Sect. 15.3.

In the global property setting, inheritance relationships between properties have similar consequences. A subproperty can define more restrictive domain and ranges than its superproperty. Inheritance also enables a property to inherit all pairs of instances connected with its subproperties. That means if two instances, *x and y*, are connected with the property u and u is a subproperty of z, then x and y are also connected with z.

For example, this is also how property inheritance in RDF Schema (RDFS) works. A subproperty can define new domain and range that are more restrictive than the domain and range of the superproperty with new types. Unlike schema.org, in RDFS, a new type added to the domain and/or range of a subproperty restricts the domain and ranges. This is because the domain and range are defined as a conjunction of the types used in their definition.

15.2.4 Instantiation Relationship

The instantiation relationship in schema.org defines to which types an instance belongs. Schema.org types can be instantiated by the users of the ontology; for example, an instance called *Oktoberfest* can be defined as an instance of the type *schema:Event*, and the *schema:Date* type can be instantiated with the literal *2021-08-01*.

Instances can have relationships with other instances as property values. For example, *Oktoberfest* can have *Munich* as the value of the *location* property.

Types more specific than *schema:Enumeration* can have a set of instances already as part of the ontology. The *schema:DayOfWeek* enumeration type has *schema:Monday*, *schema:Tuesday*, *schema:Wednesday*, *schema:Thursday*, *schema:Friday*, *schema:Saturday*, *schema:Sunday*, and *schema:PublicHolidays* as instances. Further instances of enumeration types can be defined by users like for any other type.

15.2.5 Multityped Entities

When creating a knowledge graph with schema.org, sometimes it may feel like some types would have been better to be a subtype of another type at the conceptual level or that types are missing. A very common scenario for this is offers of hotel rooms. An offer can have an item that is either a *schema:Product* or a *schema:Service* instance. This is provided by the *schema:itemOffered* property that has *schema:Product* or *schema:Service* as a range definition. However, the *schema:HotelRoom* type is not a subclass of either of them; therefore, it cannot be directly used as the value for *schema:itemOffered*.

A quick solution would be to fix this at the conceptual level by introducing *schema:HotelRoom* as a subtype of *schema:Product* or *schema:Service*. However, in the long term, this may not be a proper solution. It would require every concept that represents a commercial value for some use case to become a subclass of *schema:Product* or *schema:Service*. Making a type a subtype of another type comes with a huge commitment as it would imply both the inheritance of properties and instances. Such a commitment may not be suitable for most use cases.

Schema.org fixes this issue at the epistemological level by allowing multityped entities. This means an instance of the type *schema:Product* can also be instantiated by the *schema:HotelRoom* type. This allows the usage of properties from both *schema:Product* and *schema:HotelRoom* types. With this concept, any type can be combined during instantiation.

However, multityped entities also create an important drawback. A multitype instance is essentially an instance of the conjunction of multiple types that is defined anonymously. This hurts the explicit nature of ontologies, i.e., it is not visible at the conceptual level that there are implicit links between different types. We cannot use the ontology without an external “guide” to make implicit conceptualization

decisions explicit. This is exactly the case for hotel rooms; external documentation is provided to explain how the *HotelRoom-Product* modeling pattern should be used.[3]

15.3 Schema.org at the Logical Level

At the logical level, we will show how schema.org can be formalized by a logical formalism. We will give examples of two different formalisms:

- First-order logic (FOL) for a global property-oriented perspective, and
- A language like frame logic (F-logic) (Kifer et al. 1995) for a local property-based perspective

The structure of the epistemological level will be followed. We will first talk about types, followed by properties, inheritance relationships, instantiation relationships, and, finally, multityped entities.

15.3.1 Types

With FOL, types can be represented with unary predicates; for example, types *schema:Event*, *schema:LocalBusiness*, and *schema:Date* can be represented with *Event(x)*, *LocalBusiness(y)*, and *Date(z)* unary predicates. With F-logic, types are mapped to classes like *Event*, *LocalBusiness*, and *Date*.

15.3.2 Properties

With FOL, properties can be represented with binary predicates; for example, properties *schema:location*, *schema:name*, and *schema:startDate* can be represented with *location(x,y)*, *name(x,y)*, and *startDate(x,y)* binary predicates, where *x* and *y* are variable symbols. Property-value assertions (property instantiation) are done by value assignment to the variables, for example, *location(Oktoberfest, Munich)*, where *Oktoberfest* and *Munich* are *constants*.

Domain and range restrictions can be defined as implications from binary predicates to the conjunction of the disjunction of unary predicates. For example, the domain and range of the location property can be defined, as shown in Fig. 15.4.

With F-logic, properties are mapped to attributes of classes. For example, the *schema:Event* type can be defined with the attributes *location* and *name*. Property-value assertion in F-logic assigns values to an attribute. Figure 15.5 shows the

[3] https://schema.org/docs/hotels.html

$$\forall x, y(location(x, y) \rightarrow (Event(x) \vee Organization(x)) \wedge (Place(y) \vee VirtualLocation(y))).$$

Fig. 15.4 Definition of domain and range of the location property in FOL

$$Event[location \Rightarrow _\#(Place\ or\ VirtualLocation), name \Rightarrow Text, startDate \Rightarrow DateTime]$$

$$Oktoberfest: Event[location \rightarrow Munich, name \rightarrow Oktoberfest, startDate \rightarrow "2022-09-25"]$$

Fig. 15.5 F-logic representation for property location and name defined on the class *Event and an example instance with property value assertions* (The *_#* symbol represents a skolem constant, which would translate to "Event has some location that is a Place or VirtualLocation." The disjunctive range definition is, although syntactically correct, semantically not strongly defined in F-logic. For example, in description logic, if the location value is explicitly stated as "not Place," then we would infer that it is a *VirtualLocation*. This is not the case in F-logic (Kifer 2005).)

definition of attributes of the class *Event* and an instantiation of an *Event*, including attribute values.

Schema.org does not have any formal semantics for domain and range definitions but only an informal definition[4]:

- "each property may have one or more types as its domains. The property may be used for instances of any of these types."
- "each property may have one or more types as its ranges. The value(s) of the property should be instances of at least one of these types."

This definition indicates that domain and ranges are disjunctive. As it is not unambiguous what modal verbs like "may" and "should" mean, we can have two interpretations for how domain and range definitions are treated: constraint semantics (de Bruijn et al. 2005) and restriction semantics (Patel-Schneider and Horrocks 2007).

Constraint semantics verifies values against the range to see if a range constraint is violated (closed-world assumption). This is how F-logic deals with domains and ranges. For instance, the definition in Fig. 15.5 states that an instance of an *Event* that uses the *location* property must have as property value an instance of the type *Place* or *VirtualLocation*; otherwise, it causes an inconsistency.

Restriction semantics infers the types of instances based on domain and range definitions (open-world assumption). This is how standard FOL semantics handle domain and range definitions. The *location* predicate shown in Fig. 15.4 is also a good example of how FOL implements the open-world assumption. The domain and

[4] https://schema.org/docs/datamodel.html

$$\forall x\big(SportsEvent(x) \rightarrow Event(x)\big) \quad (1)$$
$$\forall x, y\big(sportsLocation(x, y) \rightarrow location(x, y)\big) \quad (2)$$

Fig. 15.6 FOL representation of type and property inheritance

range of the *location* property are not constrained to certain types, but the types are inferred for the assigned values of variables *x* and *y*.

15.3.3 Inheritance

With FOL, the inheritance relationship can be mapped to implication between predicates with universal quantification. Figure 15.6 shows type and property inheritance at the logical level with FOL.

The first statement in Fig. 15.6 shows type inheritance, which means every *schema:SportsEvent* instance is also a *schema:Event* instance. The second statement shows property inheritance, which means that for all *x* and *y*, if *x* and *y* are connected with the property *schema:sportsLocation*, then they are also connected with the property *schema:location*.

With F-logic, the subclass relationship is provided by the "::" operator, for example, *SportsEvent*::*Event*. In this case, *schema:SportsEvent* inherits all properties of Event as well as their domain and range definitions. The issues mentioned at the epistemological level manifest here. Disjunctive ranges in the frame-based perspective would cause issues with inheritance. This implies that an instance of a subclass may be forbidden as an instance of its superclass. Consider the following scenario: a *schema:SportsEvent* class is defined as a subclass of *schema:Event* and inherits the *schema:location* attribute from *schema:Event* with range *schema:Place*. Assume we add a new type to the range of *schema:location* on *schema:SportsEvent* such as *schema:PostalAddress*. This would imply that any *schema:SportsEvent* instance that has a *schema:PostalAddress* instance as a value for the *schema:location* attribute can be a valid instance. However, the *schema:SportsEvent* instance cannot be an instance of schema:Event as the range of the *schema:location* property on schema: Event would not accept *schema:PostalAddress* instances.

Schema.org is supposedly based on the RDFS model; however, due to the issues mentioned above, it violates the set-based semantics of inheritance (a subclass should be a subset of its superclass). Therefore, schema.org has neither proper mathematical semantics, nor does it fully use RDFS semantically (but only syntactically).

15.3.4 Instantiation

With FOL, instantiation relationships for types can be mapped to value assignment to unary predicates such as *Event(Oktoberfest)*, *LocalBusiness(DiePizzeria)*, and *Date(20211224)*, where *Oktoberfest*, *DizPizzeria*, and *20211224* are objects in the domain.

With F-logic, instantiation is done with the ":" operator. For example, *Oktoberfest:Event*, *DiePizzerei:LocalBusiness*, and *20211224:Date.*

15.3.5 Multityped Entities

Multityped entities are represented in FOL as a conjunction of two unary predicates. Figure 15.7 shows examples of multityped entities in FOL and F-logic for describing the conjunction of *schema:HotelRoom* and *schema:Product* types.

Note that the predicate *HotelRoomProduct* in FOL and class *HotelRoomProduct* in F-logic do not correspond to any explicit type in schema.org, but they are ad hoc generated types based on the conjunction of two types.

15.4 Schema.org at the Implementation Level

Schema.org uses RDF syntax and most of the RDFS primitives as syntax:

- *rdfs:Class* for types
- *rdfs:subClassOf* for type hierarchy
- *rdfs:subPropertyOf* for property hierarchy
- *rdf:type* for instantiation

It uses its own primitives for domain and range definitions, *schema:domainIncludes* and *schema:rangeIncludes*, respectively: They encode disjunctive domain and range definitions as opposed to conjunctive *rdfs:domain* and *rdfs:range* definitions.

The interpretation of these primitives is not formally defined, and their intended meaning is not necessarily compliant with RDFS. The usage of RDFS is mostly

FOL:

$\forall x((HotelRoom(x) \wedge Product(x)) \rightarrow HotelRoomProduct(x))$.

F-Logic:

$X{:}\,HotelRoomProduct :- X{:}\,HotelRoom \wedge X{:}\,Product$

Fig. 15.7 Multityped entity representation with FOL and F-logic

```
@prefix schema: <https://schema.org/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
<https://schema.org/musicReleaseFormat> schema:rangeIncludes
schema:MusicReleaseFormatType .
schema:ElectronicsStore
  rdfs:subClassOf schema:Store ;
  a rdfs:Class .
schema:colleague rdfs:label "colleague" .
schema:bankAccountType schema:domainIncludes schema:BankAccount .
schema:ParentAudience rdfs:label "ParentAudience" .

schema:DrugPregnancyCategory rdfs:label "DrugPregnancyCategory" .
```

Fig. 15.8 An excerpt from schema.org encoded with RDF(S) in Turtle syntax

syntactic. For example, subproperties do not necessarily imply their super properties, or schema.org datatypes do not directly map to XSD datatypes used in RDFS. Figure 15.8 shows an excerpt from schema.org encoded in RDFS in Turtle syntax. Any RDF serialization format can be used to encode schema.org.

15.5 Summary

Epistemologically, schema.org supports types, properties, inheritance, and instantiation. However, it does not have any formal definitions for how these primitives are represented at the logical layer.

Based on the definitions made in the documentation of schema.org, we presented two alternative logical formalisms as potential underlying semantics. FOL requires the mapping of epistemological primitives to logical primitives, for example, types and properties to unary and binary predicates. F-logic itself contains primitives like classes and attributes, so it *merges* logical and epistemological levels. This has the advantage that primitives like classes and attributes have direct model-theoretic semantics. F-logic does not need to be mapped to another logical language like first-order logic (i.e., there is no operational semantics for a mapping process involved) (Angele et al. 2009).

A significant feature of schema.org is that the properties have disjunctive ranges. Subtypes can extend the ranges of properties (like XML (Klein et al. 2001)). This causes issues for inheritance, particularly in the frame-oriented scenario. Schema.org mostly uses RDFS primitives to represent its schema. Schema.org does not use domain and range definitions from RDFS as RDFS interprets them conjunctively but introduces its own language primitives.

Schema.org provides more of a set of loose guidelines than a formal ontology. Many aspects are simply ignored. For example, subproperties are not formally

subproperties. The *schema:identifier* property is intended to have unique values, which require the introduction of functional properties in the logical layer. However, it is treated as an ordinary property that maps to a binary predicate in FOL.

On one hand, the modeling approach of schema.org is formally problematic; on the other hand, this approach is not a bug but a feature. The idea is to put as little constraints as possible on data publishers to enable them to publish as much as possible. The consumers of the data need to put effort into interpreting the data described with schema.org.

References

Angele J, Kifer M, Lausen G (2009) Ontologies in F-logic. In: Staab S, Studer R (eds) Handbook on ontologies. Springer, pp 45–70

De Bruijn J, Lara R, Polleres A, Fensel D (2005) OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic web. In: Proceedings of the 14th International Conference on World Wide Web, Chiba, Japan, May 10–14, pp 623–632

Kifer M (2005) Rules and ontologies in F-logic. In: Reasoning web: First International Summer School 2005, Msida, Malta, July 25–29, 2005, Revised Lectures, Springer, pp 22–34

Kifer M, Lausen G, Wu J (1995) Logical foundations of object-oriented and frame-based languages. J ACM 42(4):741–843

Klein M, Fensel D, Van Harmelen F, Horrocks I (2001) The relation between ontologies and XML schemas. Electron Trans Artif Intell 6(4)

Patel-Schneider PF (2014) Analyzing schema.org. In: The Semantic web – ISWC 2014: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19–23, 2014. Proceedings, Part I 13. Springer, pp 261–276

Patel-Schneider PF, Horrocks I (2007) A comparison of two modelling paradigms in the semantic web. J Web Semant 5(4):240–250

Chapter 16
Summary



Semantics deals with the interpretation of symbols to determine their meaning. Knowledge graphs as "graphs" inherently do not have any semantics. However, as a knowledge representation formalism, knowledge graphs can provide two different kinds of semantics. First, knowledge graphs can reflect human understanding, where syntactic structures represented by nodes and edges are mapped to concepts and relations in a domain. Second, the nodes and edges can be interpreted with an underlying logical formalism that defines the meaning in a machine-understandable way by mathematical means.

In Part II of the book, we introduced semantics in the framework of five levels of knowledge representation defined by Brachman (1979). These are linguistic, conceptual, epistemological, logical, and implementation levels:

- At the *linguistic level*, knowledge is represented with natural language and typically processed with Natural Language Understanding techniques. We discuss this layer in more detail in Chap. 18.
- The *conceptual level* deals with "human semantics," where a part of the world is modeled via the primitives provided by the epistemological level. Here, we think about, for example, what the concepts are in our domain and what properties they carry.
- The *epistemological level* defines the primitives (e.g., concepts, properties, inheritance, and instantiation relationships) for knowledge representation that links the logical level (e.g., predicates with different arity, functions) and conceptual level (e.g., the concept of *Person*, its properties such as *name* and *birthdate*, a subconcept of *Person* such as *Employee* and a specific instance of *Employee*).
- The *logical level* deals with the formal representation of knowledge via logical primitives. For example, classes and properties, as defined under the epistemological level, can be represented as unary and binary predicates.
- Finally, the *implementation level* deals with how the knowledge is encoded in a computer.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_16

Modeling for knowledge graphs typically starts at the conceptual level and most prominently is achieved with the help of ontologies, which we have covered in Sect. 2.5. We will further go into detail in Chap. 18. In this part, we focused on the epistemological and logical levels.

There are many languages that are commonly used for modeling and manipulating knowledge graphs. When we talked about epistemology, we introduced the main features and primitives these languages provided, namely, Resource Description Framework (RDF) and RDF Schema (RDFS) for modeling a knowledge graph, SPARQL Protocol and RDF Query Language (SPARQL) and Shapes Constraint Language (SHACL) for querying and verifying it, and Web Ontology Language (OWL) and rules for reasoning about knowledge. Additionally, we introduced the Simple Knowledge Organization System (SKOS) as a lightweight approach for integrating the formal world of ontologies with less formal ways of knowledge organization systems.

An important feature of knowledge graphs is that they provide a declarative way to make implicit knowledge explicit. This is only possible if the languages used to model a knowledge graph also provide formal semantics, which is typically achieved via logical foundations. When we talked about logic, we introduced the fundamentals of logic and the formal semantics based on these fundamentals for languages, like RDF, RDFS, OWL, and rules. While RDF(S) has its own flavor of model-theoretic semantics, OWL and rules use subsets of first-order logic (FOL) such as description logic or FOL with minimal model semantics. We also described formal definitions for the SPARQL query language based on an algebra that describes how query engines should interpret SPARQL queries.

To make understanding the five levels of knowledge representation and semantics more concrete, we provided an analysis of a widespread ontology called schema.org, which lies at the conceptual level, and how it can be mapped to different logical formalisms via the epistemological level. More interestingly, we examined the consequences of the decisions made at the epistemological level for the conceptual and logical levels.

Part II was meant to provide the theoretical foundations to truly understand knowledge graphs and the operations that can be taken on them. Many of the topics covered by this section will have practical applications in the upcoming Part III, where we will show how to create, host, assess, clean, enrich, and deploy a knowledge graph.

Part III
Knowledge Modeling

Chapter 17
Introduction: The Overall Model



Knowledge graphs are not built once and then forgotten, but they are "living" entities that need to be created and maintained as useful resources for applications (Tamasauskaite and Groth 2023). Starting with this chapter, we will introduce different steps of the life cycle of our knowledge graph development approach. We will introduce an overall process and task model (Fensel et al. 2020). For each task, we will talk about the following:

- What are the main challenges?
- What are the approaches to tackling those challenges?
- What is the available tool support?

Figure 17.1 provides the overall *process model* of our knowledge graph development approach. There is an initial effort to build and host a knowledge graph. Data need to be collected from various sources and aligned. We call this process *knowledge creation*. It is obviously an essential task because if there is no knowledge, there is no power. These data must be hosted. Obviously, the graph model and the size of these data put strong requirements on potential solutions for *knowledge hosting*. We need to provide support in reliably storing and efficiently querying such data.

After this process, we enter the *knowledge curation* cycle, i.e., knowledge graph maintenance:

- *Knowledge assessment* evaluates the quality of the built knowledge graph. Various dimensions define the quality of a knowledge graph that will be discussed. The result can be to provide the knowledge graph for deployment or, more likely, to identify the need for correcting and/or extending it.
- *Knowledge cleaning* is about identifying errors in the knowledge graph and correcting them.
- *Knowledge enrichment* is about identifying gaps in the knowledge graphs as well as trying to fill them. Additional data sources may be integrated for this purpose.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_17

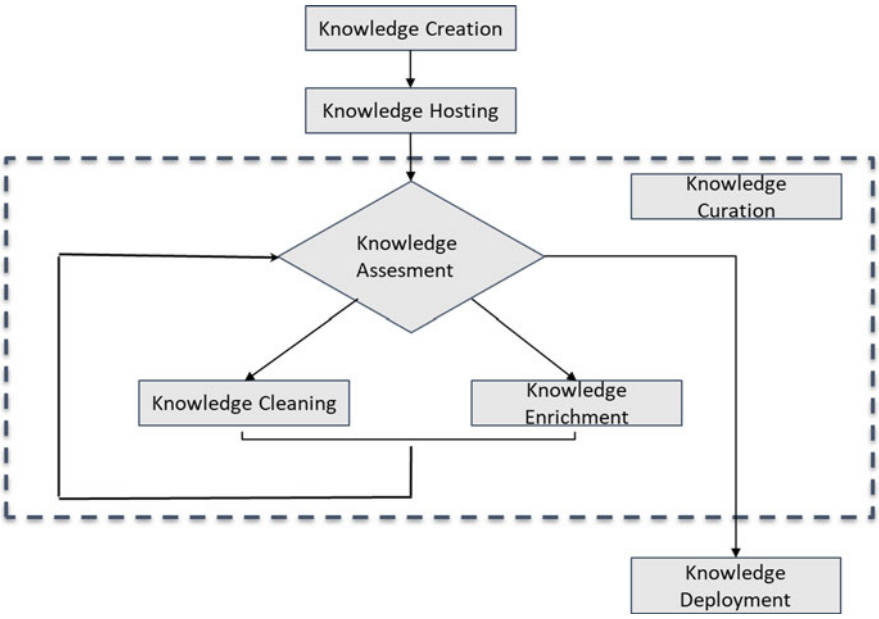


Fig. 17.1 The process model for building and maintaining knowledge graphs (Fensel et al. 2020)

- *Knowledge deployment* is about making a knowledge graph available for an application that uses it to provide better services (like a knowledgeable chatbot). The concept of views is essential for it.

Figure 17.2 provides a more refined task model, which will be discussed in the following chapters:

- Chapter 18 discusses approaches for creating a knowledge graph.
- Chapter 19 discusses approaches for hosting a knowledge graph.
- Chapter 20 discusses approaches for evaluating the quality of a knowledge graph.
- Chapter 21 discusses approaches for error detection and correction, i.e., knowledge cleaning.
- Chapter 22 discusses approaches for extending the coverage of a knowledge graph by additional facts, i.e., knowledge enrichment.
- Chapter 23 discusses concepts and tools for deploying knowledge graphs in working environments, i.e., knowledge deployment.
- Finally, Chap. 24 provides a summary of Part III.

This will give us a complete and detailed picture of the knowledge graph building and maintenance process.

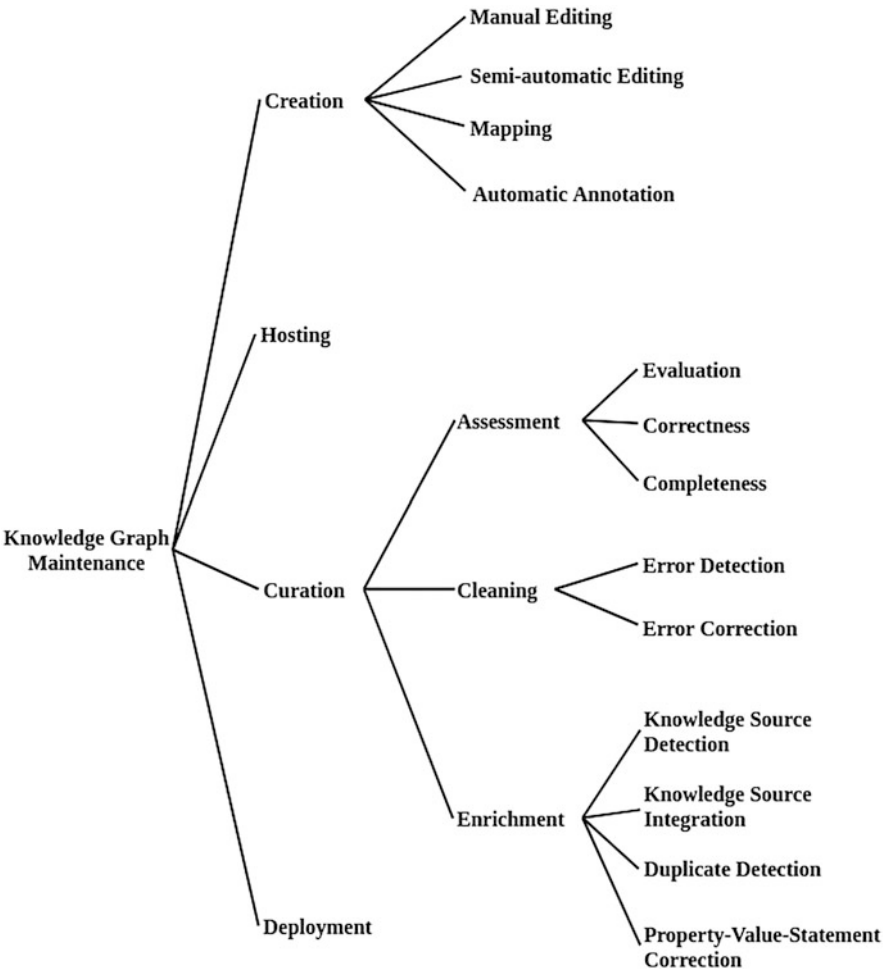


Fig. 17.2 The task model for building and maintaining knowledge graphs (Fensel et al. 2020)

References

Fensel D, Simsek U, Angele K, Huaman E, Kärle E, Panasiuk O, Toma I, Umbrich J, Wahler A (2020) Knowledge graphs. Springer

Tamasauskaite G, Groth P (2023) Defining a knowledge graph development process through a systematic review. ACM Trans Softw Eng Methodol 32(1). https://doi.org/10.1145/3522586

Chapter 18
Knowledge Creation



In Part I, we introduced the difference between a TBox and an ABox. Generating a TBox (an ontology) and an ABox is quite a different activity. A TBox captures terminological knowledge that provides classification, properties, constraints, and inference rules for data. When meant to be reusable and not just built for a specific application, it is an ontology. An ABox contains the actual data, in our case, semantically annotated by the vocabulary provided by the chosen TBox.

We will first briefly cover the ontology creation process and then focus on ABox generation since knowledge graphs are typically more ABox heavy. We will also see that both areas partially overlap, mainly when an ontology is derived or adapted by a given set of data.

18.1 Ontologies

What is an ontology? Different definitions emerged over the years, for example:

- An ontology defines the basic terms and relations comprising the vocabulary of a topic area and the rules for combining terms and relations to define extensions to the vocabulary (Neches et al. 1991).
- An ontology is an explicit specification of a conceptualization (Gruber 1993).
- An ontology is a hierarchically structured set of terms describing a domain that can be used as a skeletal foundation for a knowledge base (Swartout et al. 1996).
- An ontology provides the means for describing explicitly the conceptualization behind the knowledge represented in a knowledge base (Bernaras et al. 1996).

The definition from Studer et al. (1998) merges the key aspects from these definitions, as shown in Fig. 18.1. Features of an ontology are that they model knowledge about a specific aspect, define a common vocabulary, and define how concepts are interrelated. The meaning of terms should rely on some formal logic. It

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_18

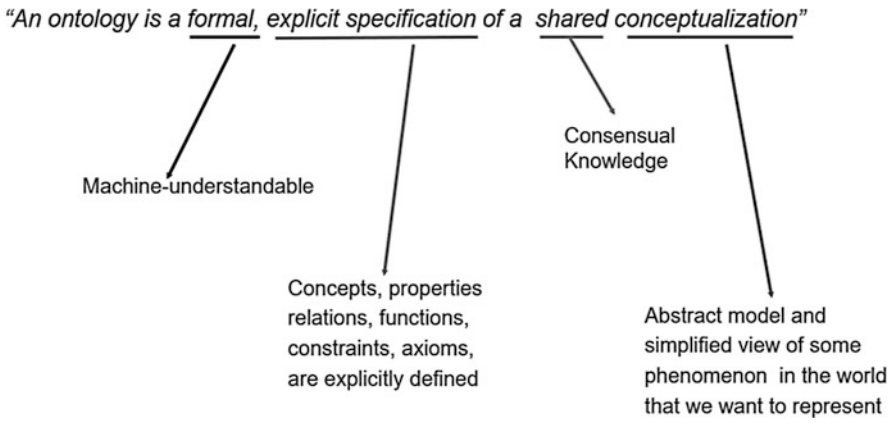


Fig. 18.1 The definition of an ontology (see Studer et al. (1998))

consists of a conceptualization and often of an implementation. It is built up by ontological primitives.

In the following, we will provide some examples of ontologies. We split them into two categories:

- *Domain-independent ontologies*: These ontologies provide foundational concepts and relationships that are not specific to a domain. They are commonly known as top-level, upper-level, or foundational ontologies.
- *Domain ontologies*: These ontologies provide terminology specific to a given domain, for example, concepts like *PathologicalProcess* in clinical trials domain. Domain ontologies may base themselves on domain-independent ontologies to improve interoperability. For example, *PathologicalProcess* in a domain ontology may be defined as a subclass of the *Process* concept in a foundational ontology.

18.1.1 Domain-Independent Ontologies

Domain-independent (upper-level) ontologies are not designed to describe a specific domain but should rather provide high-level terms to describe various domains at a unified level.[1] We will cover some examples of such ontologies briefly.

The BFO (Basic Formal Ontology)[2] is a small upper-level ontology that aims to facilitate data integration and information retrieval tasks across many domains. It contains a total of 35 classes, where 34 classes are specialized from a type called

[1] See also https://en.wikipedia.org/wiki/Upper_ontology

[2] http://basic-formal-ontology.org/

Entity. It is mainly used for scientific domains. More than 250 domain ontologies use BFO. It is primarily adopted by the biomedicine domain.

The ontology of the Cyc knowledge graph contains mainly common-sense knowledge: "Causes start at or before the time that their effects start." It is modeled with the CycL language (more complex than first-order logic); see Lenat and Guha (1989). It is organized through microtheories (Guha 1991), which provide a way to decompose the ontology in hierarchically organized subsets to avoid inconsistencies that may happen in a large knowledge base.

DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) (Masolo et al. 2003) aims to cover the terminology for natural language and human common sense. It has been used by domain ontologies for domains like law, biomedicine, and agriculture.[3]

18.1.2 Domain Ontologies

Domain ontologies offer classes and relationships to model-specific domains. Naturally, there are a vast number of domain ontologies, which would be impossible to cover here. In the following, we will present four examples from different domains.

The Material Design Ontology (MDO) (Li et al. 2020) is an ontology that provides classes and relationships to model the material design domain, mainly from the solid-state physics perspective. It is based on various ontologies, including the European Materials Modeling Ontology (EMMO),[4] an ontology for material science (Goldbeck et al. 2019). MDO is using OWL 2 DL as formalism.

Dublin Core is an RDFS Schema (RDFS) ontology for metadata management for electronic resources. It is commonly used in the digital and printed publication domain for creative works. It contains concepts like *BibliographicResource* and *Dataset*. . . and properties like author, license, language, audience. . .[5]

eClassOWL (Hepp 2005) is an ontology for product classification (modeled with RDFS/Web Ontology Language (OWL)).[6] It is based on the widely adopted product and service classification standard eCl@ss.[7]

Schema.org[8] was started in 2011 by Bing, Google, Yahoo!, and Yandex to provide vocabularies for annotating Web sites. It covers multiple domains that have commercial value on the Web, e.g., events, accommodation, and points of interest. Meanwhile, it has become a de facto standard for annotating content and data on the Web with around 802 types, 1464 properties, 14 datatypes,

[3] http://www.loa.istc.cnr.it/dolce/overview.html

[4] https://matportal.org/ontologies/EMMO

[5] https://www.dublincore.org/specifications/dublin-core/dcmi-terms

[6] http://www.heppnetz.de/projects/eclassowl/

[7] https://eclass.eu/

[8] https://schema.org

87 enumerations, and 463 enumeration members (in May 2023). It is based on a variation of RDFS.

Note that it may not always be clear what the line between domain independent and domain ontology is. In fact, some practitioners and researchers also consider middle-level ontologies as a transition layer between upper-level and domain ontologies. Such ontologies typically cover multiple aspects of various domains while being more specific than upper-level ontologies. For example, the European Materials Modelling Ontology (EMMO) can be seen as such an ontology.

18.1.3 Ontology Engineering and Methodologies

Ontology engineering is defined as "the activities that concern the Ontology development process, the ontology life cycle, and the methodologies, tools, and languages for building ontologies" (Gomez-Perez et al. 2006).

Much like software engineering methodologies, ontology development methodologies consist of certain steps supported by guidelines. Their main contribution is to provide a well-structured way of developing ontologies instead of ad hoc approaches based on intuition, with low reusability and verifiability. Methodologies provide guidelines on how the ontology-building process should be structured. Ontology engineering methodologies may support a combination of different development paradigms, such as collaborative and iterative development. Examples of such methodologies are:

- Uschold and King (1995)
- Gruninger and Fox (1995)
- METHONTOLOGY (Fernández-López et al. 1997)
- On-To-Knowledge Methodology (Sure et al. 2004)
- Ontology Development 101 (Noy and McGuinness 2001)
- DILIGENT (Pinto et al. 2004)
- HCOME (Kotis et al. 2005)
- UPON (De Nicola et al. 2005)
- NeOn Methodology (Suarez-Figueroa 2007)
- LOT Methodology (Poveda-Villalon et al. 2022)

We provide *Ontology Development 101* as an illustration as it covers the essence of many other ontology development methodologies. The core of the methodology consists of seven steps (see Noy and McGuinness (2001)):

1. Determine the scope of the ontology.
2. Reuse existing ontologies.
3. Identify relevant terminology.
4. Define classes.
5. Define properties.

6. Define (domain) and range constraints.
7. Populate the ontology with exemplary instances.

The first step is to *determine the scope of the ontology*. This includes identifying the domain of the ontology and its target use cases, as well as its stakeholders. A core tool for this step is to use competency questions. These questions are formed according to the requirements specified by domain experts and applications and are typically formalized as queries for the underlying ontology (e.g., SPARQL Protocol and RDF Query Language (SPARQL) queries for RDFS/OWL ontologies).

After the scope is identified, we continue with the consideration of ontology reuse. The main idea of this step is to identify existing ontologies that may be relevant to our domain and scope. Alongside relevancy, other criteria (Lourdusamy and John 2018) can be adopted for determining ontologies to reuse, such as various quality metrics (e.g., conciseness, completeness, formal correctness, following naming conventions)

The next step is to *identify important terms for our ontology*. At this stage, we do not necessarily make any distinction between classes and properties but just create terminology addressing the needs of our domain.

After establishing the terminology, we *define classes and the class hierarchy*. Typically, the nouns identified in the previous step are good candidates to be classes. A class denotes a set of instances that share common features. The class hierarchy can be defined as top-down, bottom-up, or middle-out. The top-down approach starts with the definition of most general classes and specializes them. The bottom-up approach starts with the most specific classes and tries to find common superclasses to group these classes. Finally, the middle-out approach starts with the most prominent classes and moves in both specialization and generalization directions.

After determining the classes and their hierarchical organization, it is now time to *define the properties* in our ontology. Properties are used to describe certain characteristics of instances belonging to certain classes. Typically, what is left from the terminology after defining the classes are properties. As a rule of thumb, the verbs in the terminology are typically good candidates for being properties. Properties are inherited top-down across the class hierarchy, and they should be defined for the most general class possible.[9]

The defined properties can be *further refined with constraints*. Depending on the modeling paradigm, these constraints can be domain and range constraints. For example, in the frame-based approach, the domain of a property is already determined at the previous step since properties are defined on classes. With a property-oriented modeling paradigm like RDF(S) uses, domains are defined directly on properties. Range constraints further restrict the type of values a property can have. Depending on the modeling language, the range constraints may have different expressive power (e.g., restricting the type of values, cardinality, range for numerical values).

[9] Some readers may have already realized the top-down inheritance of properties is more accurate for the frame-oriented approach.

Finally, we can create some instances to exemplify our ontology. Although instances typically belong to ABox, a small number of instances can be created as part of the ontology, for example, as enumeration members and to demonstrate how the classes can be instantiated in the ABox.

18.1.4 Ontology Manifoldness

Up to now, we have been focusing on the case of having one ontology. However, there are several reasons that lead to several ontologies that need to be considered in parallel. There are the following approaches to deal with this need:

- Ontology modularization
- Ontology alignment
- Ontology merging, and
- Ontology networks

We will discuss them in the following.[10]

18.1.4.1 Ontology Modularization

Large ontologies may be decomposed for the better handling of the formal and modeling aspects of knowledge-based systems. For example, *CommonKADS* (Schreiber et al. 2000), a methodology for developing knowledge-based systems, decomposes ontologies into models like task, domain, and organization in order to enable the separation of concerns and distributed development. It is decomposition based on different aspects.

Cyc (Lenat and Guha 1989) contains many assertions for mostly common-sense knowledge. For structuring this plethora of assertions, it uses the notion of microtheories (Guha 1991). Microtheories are subsets of a knowledge base that contain axioms and assertions about specific parts of the modeled world. Each microtheory has a "theme," that is, the assertions come from a common source or they have common assumptions about the modeled world. The assertions in a microtheory are consistent but not necessarily consistent across different microtheories.

18.1.4.2 Ontology Alignment

One of the major motivations of the Semantic Web is to enable data integration on a Web scale with the help of ontologies. However, it would be unrealistic to expect

[10] Sections 18.1.4.2 and 18.1.4.3 are mainly based on Sack (2015).

one true "world ontology." Due to the subjective nature of ontologies in regard to domains and tasks, there can be heterogeneities at the instance and schema levels. Ontology alignment aims to resolve these heterogeneities between different ontologies. Ontologies may heterogeneously describe the overlapping parts of the world in the following ways:

- The same syntactical structures describe different notions in the modeled world.
  - For example, a *Player* can refer to someone who plays video games or to someone who does some sport, e.g., a football player.
- Different syntactical structures describe the same notion in the modeled world.
  - For example, synonyms or words with the same meaning in different languages or different modeling languages are used.
- Different modeling conventions and paradigms are used.
  - For example, one ontology may model a path and its length between two locations as an n-ary relation, and another one may leave it to the ABox, where the distance between two locations is attached to a triple specifying a path between two locations with the help of reification.
- The granularity of the conceptual modeling varies.
  - For example, one ontology may just model the *Player* type, but the other one does more fine-grained modeling by defining subtypes of the *Player* type, such as *FootballPlayer* and *VolleyballPlayer*.
- Finally, different stakeholders may have different points of view.
  - For example, some countries allow dual citizenship, while others do not.

Different kinds of heterogeneities can be solved with different levels of difficulty. For example, syntactic heterogeneities such as using different RDF serialization formats can be solved rather easily without sacrificing the semantics; however, solving heterogeneities at the conceptual level may be more challenging. Strongly defined concepts and different points of view may cause logical side effects when they are aligned at the conceptual level, e.g., the concept representing a *Person* in different ontologies may cause a logical inconsistency if they are aligned when they contain different numbers of cardinality restrictions on a property, like *hasCitizenship*. See also Sect. 22.3 for different ontology alignment techniques.

18.1.4.3 Ontology Merging

Ontology merging is strongly related to ontology alignment, with a subtle difference. Ontology alignment does not necessarily imply the creation of a new ontology. However, in ontology merging, two initial ontologies are combined into one

ontology. There are two main strategies for ontology merging, namely, the union and intersection approaches:

- The *union approach* creates a set union of all terms of the initial ontologies. This approach may cause conceptual and logical inconsistencies that need to be resolved.
- The *intersection approach* only uses the terms from the initial ontologies that overlap in their definitions. This approach is less likely to cause inconsistencies, but it may sacrifice coverage and granularity.

18.1.4.4 Ontology Networks

We have already discussed the alignment and merging of two ontologies. It is necessary due to heterogeneities among ontologies. Applying ontologies on a distributed system like the Web brings the alignment issue to a whole different level. It is hardly ever possible to expect different communities that have an interest in the same or different topic to have the same understanding. Also, there is no central authority on the Web that could enforce such a global view. We rather encounter the emergence of "different islands of meaning." Small groups can commit to a detailed conceptualization of a part of the world more easily than larger groups. Knowledge exchange among these islands is only possible if:

- The ontologies used in these islands align themselves with more abstract ontologies with weaker ontological commitments (e.g., upper-level ontologies or less concrete domain ontologies) or
- They use an alignment process with a related ontology and automatically get plugged into a network of other already aligned ontologies

Such multiple alignment of ontologies form a network structure (see Fensel et al. (1997), Benjamins et al. (1999)). One could assume that such a network is actually provided by Linked Open Data. Although they can be seen as quite related, ontology networks and Linked Open Data have fundamental differences. They are both involved with the Web-scale linkage of knowledge. However, ontology networks deal with terminological knowledge, whereas Linked Open Data is about linking instances between two datasets. Essentially, LOD is mostly about describing instances with a mixture of types and properties from different ontologies (fostering reuse) and linking seemingly related instances between datasets (mostly with owl: sameAs links). Thus, a network of ontologies links ontologies, while Linked Open Data connects data sources primarily at the ABox level and may use various ontologies to annotate them.

Actually, the *Simple Knowledge Organization System (SKOS)* is a lightweight approach to align ontologies for creating ontology networks. SKOS[11] is a common

[11] See also Sect. 13.4.

data model for representing, sharing, and linking knowledge organization systems (KOS), e.g., ontologies via the Semantic Web.

18.1.5 Summary

An ontology is an explicit formalization of a shared conceptualization. It models a part of the world from a *certain point of view* and answers the following questions:

- What are the concepts?
- What are the relationships between them?
- What are the constraints?

Ontologies are the backbone of the Semantic Web and knowledge graphs. They give human- and machine-understandable meaning to content and data on the Web and hereby enable querying and inference.

Ontology engineering provides methodological approaches for building ontologies. A typical ontology engineering methodology consists of requirement specification, conceptualization, formalization, implementation, and evaluation steps: It is usually applied iteratively. From many aspects, such ontology methodologies resemble software engineering methodologies.

It is not feasible to assume that one unified ontology can exist. People have different points of view, which leads to heterogeneity among ontologies. Different ontologies must be aligned and merged. Knowledge exchange at a large scale needs some sort of agreement between different communities. Ontologies can be linked as ontology networks to improve this situation.

After covering terminological knowledge creation, we will discuss knowledge creation at the factual level in the following section.

18.2 ABox Creation

In general, we have two kinds of knowledge:

- *Terminological knowledge (TBox)*: terminology and rules typically modeled with ontologies
- *Assertional knowledge (ABox)*: factual knowledge formulated using terminological knowledge

In the following, we will discuss different approaches for generating factual knowledge, i.e., data for a knowledge graph.

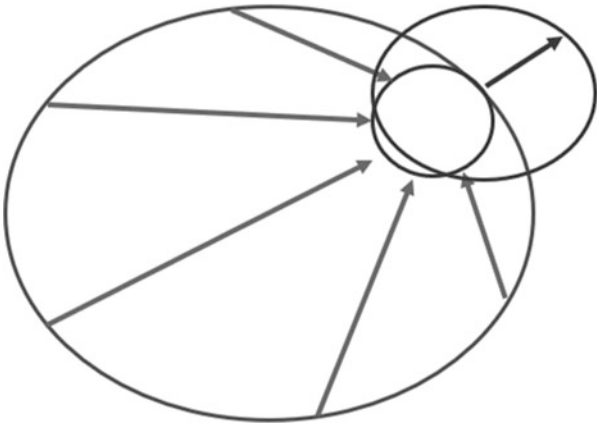


Fig. 18.2 Extension and reduction as core operations of domain specification. The small dark circle represents a subset of schema.org; the large one represents an extension of this subset

18.2.1 Domain Specification, a Connector for ABox and TBox

In this section, we focus on the creation of an ABox. We base our discussions on schema.org and use it as the golden standard. However, we apply a very lightweight ontology development process, called *domain specifications* (Fensel et al. 2020), to fix some of the issues with the usage of schema.org.[12] That is, we provide a lightweight ontology adaptation process driven by the available data and purpose. Before we generate knowledge, we need to answer the following questions:

- What entities and relationships exist in the source?
- What types and properties do they correspond to in schema.org?

The major issue with answering these questions is the nature of schema.org:

- Schema.org covers many domains and contains more than 1400 properties and more than 800 types. This makes it hard to find the right types and properties for a certain domain. Often, the provided vocabulary is much too rich and broad to annotate a certain domain.
- The domain coverage is shallow. For a given domain, specific types are usually missing. Often, the provided vocabulary is much too shallow and poor to annotate a certain domain.

This is a consequence of the purpose of schema.org. It should provide vocabulary for annotating content and data sources as vast and heterogeneous as the Web.

So we will discuss the process of ontology adaptation for a specific domain and tasks, i.e., an important aspect of ontology engineering. The approach we introduce is called *domain specification*. The two important operations are *reduction* and *extension* (see Fig. 18.2).

[12] https://schema.org/

Reduction is necessary because of the large size of schema.org: The large size makes it harder to pick the right types and properties for a specific domain. Reduction happens roughly as follows: eliminate the types and properties that are irrelevant to a given domain. For example, for a domain-like accommodation in Europe, the property for the North American Industry Classification System number (*schema:naics*) may not be relevant. You can restrict the ranges of remaining properties defined on the remaining types by removing a type from the range or replacing a type in the range with a subtype. Optionally, one can add constraints to the properties (e.g., cardinality).

Extension is necessary because of the shallow domain-specific coverage of schema.org. Many domains are partially covered, but some domain-specific types and properties may not exist in the vocabulary, e.g., the *LodgingBusiness* type has six subtypes in schema.org, but none of them is suitable for describing a "Hotel Garni," or there is no property for the boarding type of an accommodation. Extension roughly happens as follows:

- Extend schema.org with new types and/or properties and add them to the domain specification.
- Extend the range of properties with new types.

Syntactically, the reduction and extension can be made by applying a Shapes Constraint Language (SHACL)[13] operator on the schema.org vocabulary. SHACL operators define the following for a domain:

- Types that are relevant to a domain and
- Properties that are defined on the selected types, their ranges, and further constraints (e.g., cardinality)

SHACL is a language to constrain RDF graphs. Its purpose is aligned with the conceptualization and purpose of domain specification. A possible formalism for specifying it is the use of SHACL operators; see Şimşek et al. (2020).

18.2.2 Bottom-Up and Top-Down Domain Specification

Our knowledge generation approach has a bottom-up and top-down nature that takes domain specification modeling in the middle[14] (Fig. 18.7):

- *Bottom-up:* The process starts from data/content to domain specification modeling.
- *Top-down:* The process applies domain specifications to the annotation of content and data.

[13] See Sect. 13.2.2.

[14] See also Simsek et al. (2022).

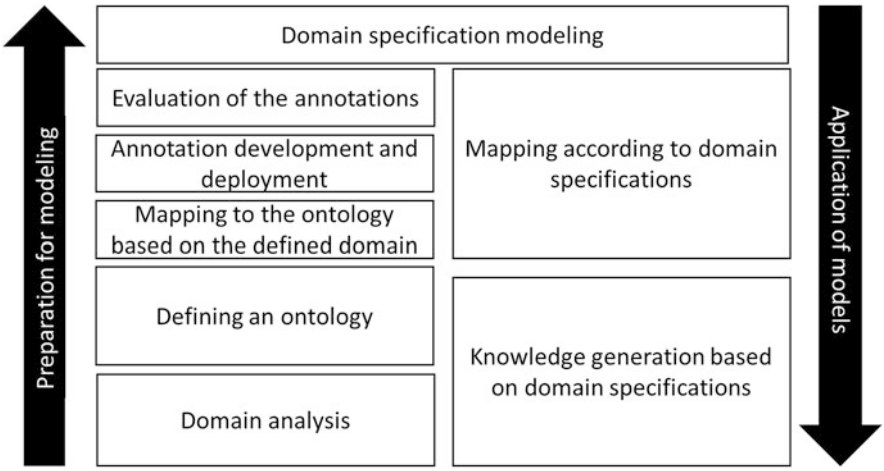


Fig. 18.3 The combined bottom-up and top-down approaches of domain specifications

18.2.2.1 Bottom-Up Development

The bottom-up domain specification process consists of the following steps; see Fig. 18.3:

1. *Domain analysis*: It is the task of analyzing real-world entities in a domain and their online representation. The aim is to identify the relevant entities for domain specification. For content, this step is typically done on Web sites. For data, this is done on database schemas or Application Programming Interface (API) metadata.
2. *Defining an ontology*: This involves the task of analyzing existing ontologies to find the ones that fit the domain at hand. In our case, we use schema.org as a basis. An important part of this step is to find out what types and properties exist and what is missing in schema.org.
3. *Mapping to the ontology based on the defined domain*: At this step, based on the results of Steps 1 and 2, we create domain specifications by reducing and extending schema.org. For the annotation of content, we identify which Web pages correspond to which type and properties. For the case of data, map the metadata of the source to the types and properties specified in the domain specification.
4. *Annotation development and deployment*: Now that we have the ontology and conceptual mapping, we can develop and deploy initial annotations. For content, manual or semi-automated knowledge creation based on the domain specification can be adopted. For data, semi-automated annotation based on declarative mappings is a feasible approach.
5. *Evaluation of the annotations*: After annotations are deployed, they need to be evaluated. The evaluation methods can vary depending on the deployment target and purpose. Annotations of content can be evaluated by monitoring the

performance of Web pages in search engine results. Annotations of data can be evaluated by testing with queries that are tailored to the relevant use cases.
6. *Domain specification modeling*: After the evaluation, the final changes to the domain specification are done, and domain-specific patterns are formalized via SHACL operators.

18.2.2.2 Top-Down Application

The top-down application of modeled domain specifications consists of the following steps:

1. *Mapping according to domain specifications*: Once the domain specifications are matured, the mappings are created/updated (particularly for (semi)-structured data sources).
2. *Knowledge generation based on domain specifications*: The mappings are then used to create and deploy knowledge from incoming content and data.

Note that the bottom-up and top-down parts of the approach have a duality: the bottom-up part aims to specify and formalize a domain for knowledge creation. It also involves the creation of a small set of annotations to enable the evaluation of the model. The top-down part aims to apply the created domain specifications on a larger scale.

18.2.3 Manual, Semi-Automatic, and Automatic Creation of Knowledge

In the following, we will discuss the manual annotation creation of annotations, the semi-automated annotation creation of content, mappings for the semi-automatic annotation creation of data, and automated annotation creation.

Manuel knowledge creation is mostly suitable for static annotations in small volumes, e.g., a business and its contact information. A widely used editor for this task is Protégé;[15] however, it can also be guided by domain specifications (Kärle et al. 2017). Actually, a form interface can be generated automatically based on a domain-specific pattern.

Semi-automatic creation of knowledge is a widely adopted approach to knowledge creation from unstructured sources like text. It is typically provided by *Natural Language Processing techniques* (Clark et al. 2012; Maynard et al. 2016). Such techniques are important when large amounts of unstructured content are obtained, e.g., via Web crawling. Natural Language Processing (NLP) is a field that addresses the challenge of processing, analyzing, and understanding natural language. An NLP

[15] https://en.wikipedia.org/wiki/Prot%C3%A9g%C3%A9_(software)

application such as information extraction typically deals with three main tasks: linguistic preprocessing, named entity recognition (NER), and relation extraction, which we already discussed in Chap. 6. We introduced the *General Architecture for Text Engineering (GATE)*[16] for this process with a running example in Sect. 6.1. There are, of course, many other NLP systems used in academia and industry, for example, CoreNLP, developed by the Stanford NLP Group at Stanford University (Manning et al. 2014). It is implemented in Java and offers the entire NLP pipeline.[17]

Recently, large language models, like GPT (Generative Pretrained Transformer) (Radford et al. 2018), have gained popularity for tackling NLP tasks in an end-to-end manner. For example, OpenAI's ChatGPT[18] can extract RDF data from unstructured text given proper prompts.

Mapping-based creation of knowledge addresses the majority of knowledge creation activities in many use cases. It is based on mapping structured data sources onto the terminology of a knowledge graph. The main idea is to map the metadata of a (semi-)structured source to an ontology and populate the instances based on the source data. This can be done programmatically via wrappers. This may seem attractive at first as it gives you the power of general-purpose programming languages. However, it does not scale due to very low reusability and portability. Ideally, we use declarative mappings and a generic mapping engine. Mappings then remain easily adaptable and reusable.

Following the standardization of R2RML (RDB to RDF Mapping Language) by the World Wide Web Consortium (W3C)[19] for creating RDF data from relational databases, many declarative mapping languages were developed:

- Ontop Language (Calvanese et al. 2017) for creating virtual knowledge graph mappings
- xR2RML (Michel et al. 2017), which extends R2RML with some useful features like accessing outer fields, dynamic language tags, and (nested) RDF lists/containers
- SPARQL-Generate (Lefrançois et al. 2017), a template-based language that benefits from the expressivity of SPARQL
- ShExML (Garcia-Gonzalez et al. 2020), a language that separates the extraction and representation of data based on ShEx.

[16] https://gate.ac.uk/

[17] https://stanfordnlp.github.io/CoreNLP/

[18] https://chat.openai.com/

[19] R2RML: RDB to RDF Mapping Language- W3C Recommendation 27 September 2012.

Perhaps one of the declarative approaches that gained the most traction is RML (RDF Mapping Language)[20] (Dimou et al. 2014). It is an extension of R2RML (RDB to RDF Mapping Language).[21] It considers as data source not only relational databases but also any kind of tabular and hierarchical source, e.g., JavaScript Object Notation (JSON), Extensible Markup Language (XML), and comma-separated values (CSV). It supports a Turtle-based syntax, as well as a YAML-based syntax, called YARRML (Heyvaert et al. 2018). Figure 18.4 shows how RDF data are generated with the help of RML mappings. The heterogeneous data sources are fed into an RML mapping engine together with RML mapping specifications that map the metadata of the source to an ontology. There is a plethora of RML mapping engine implementations, such as Morph-KGC (Arenas-Guerrero et al. 2022) and SDM-RDFizer (Iglesias et al. 2020), that optimize different aspects, such as execution time and memory usage. There are also Web-oriented implementations, like *RocketRML* (Şimşek et al. 2019), that can run in a browser or as a separate NodeJS library.[22]

As a more concrete illustration of knowledge creation based on mappings, we give an example from the knowledge creation process of the German Tourism Knowledge Graph, which we will be using for examples and illustrations throughout Part III. Figure 18.5 shows example data from an IT provider about events in JSON format. The data consist of the name, description, and URL of the event for different languages, as well as the start and end dates of the event.

Figure 18.6 shows the RML mapping in YARRML syntax for the data in Fig. 18.5. There are several things to note about this mapping specification. First, property values in the resulting RDF that is not a datatype are treated as a new logical data source. For example, the values of *schema:image* property for the *schema:Event* instances are coming from another mapping, called *image*, whose logical source is defined by the JSONPath iterator *$.*.event.images.** . The *schema:ImageObject* instances generated by the image mapping are joined with the *schema:Event instances* via the JSON paths *external_id* and ^^*external_id*. This join operation assigns the correct images to the correct events based on the unique *external_id* property of events.[23] The second thing to note is the usage of functions. The functions are rather implementation specific, more precisely specific to the RocketRML mapping engine. The functions are provided to the mapping engine externally and referred to by their name in the mapping. For example, in this

[20] See https://rml.io/specs/rml/ for the latest specification. As of June 2023, the specification is under construction by the W3C Knowledge Graph Construction Community Group and a new update will appear soon (Last accessed May 2023). See also https://www.w3.org/groups/cg/kg-construct.

[21] R2RML: RDB to RDF Mapping Language- W3C Recommendation 27 September 2012.

[22] See https://github.com/kg-construct/awesome-kgc-tools for a list of knowledge creation tools.

[23] RML is based on R2RML; therefore, it relies on joins. Since the data are nested in the JSON object, the backward traversal operator ^ is used to specify the join fields in RocketRML. This is a work-around used to deal with nested objects in hierarchical formats like JSON, where the child objects do not have explicit values for join fields. See https://www.npmjs.com/package/jsonpath-plus for the ^ operator.

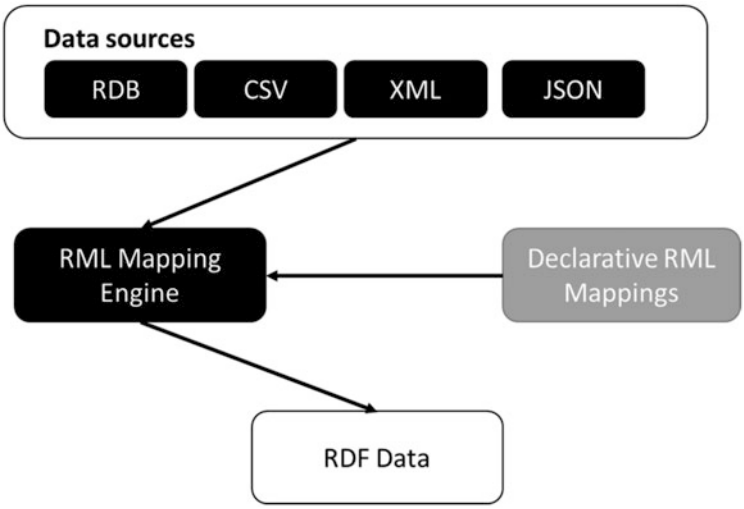


Fig. 18.4 RDF generation with the RDF mapping language (RML) (Adapted from http://semweb.mmlab.be/rml/RML_details.html)

mapping, a function called *asDateTime* is used to normalize the date and time values.

Finally, Fig. 18.7 shows the RDF data generated based on the JSON data and mapping specification in N-triples format.

Automatic creation of knowledge deals with knowledge creation with machine learning methods, ideally unsupervised. This would be the "holy grail" of knowledge creation, but usually, it comes with low accuracy, and the results are typically not explainable. Many useful approaches require human interaction or preprepared training data or trained models to some extent. For example, NELL (Mitchell et al. 2018) extracts facts automatically, but the extracted facts are reviewed, and concept clusters are labeled from an existing knowledge graph. Another example is OpenIE (Mausam 2016), which is mainly an unsupervised approach for extracting triples from an open-domain text (Fig. 18.8).[24] Machine learning approaches became popular for certain tasks for building knowledge graphs. Similarly, knowledge graphs have been used as input to improve/train machine learning applications; see Paulheim (2018).

In a nutshell, automated knowledge generation is quite useful and can be effective, particularly for working with unstructured data sources. However, it also suffers from general and task-specific limitations of machine learning. It is nearly impossible to debug when something goes wrong. The algorithms are not aware of any common sense and domain knowledge to remedy wrong statement creation (Deng

[24] Figure adapted from https://github.com/dair-iitd/OpenIE-standalone

```
[
  {
    "event": {
      "external_id": "15dc0b05-01d8-4dbe-9cf0-3f579c1fd3d7",
      "url_translations": {
        "de": "http://www.ahrtal.de",
        "en": "http://www.ahrtal.de"
    },
      "name_translations": {
        "de": "Bernd Stelter",
        "en": "Bernd Stelter"
     },
      "description_translations": {
        "de": "Hurra, ab Montag ist wieder Wochenende! heißt das neue
Programm von Bernd Stelter. Ein Programm für alle, die über Montage
mosern, über Dienstage diskutieren, die Mittwoche mies und Donnerstage
doof finden. \n„Und was machst du am Wochenende?“ – „Party!“ Es geht
um Work-Life-Balance. … Tickets online buchen"
      },
      "images": [
        {
          "order": 1,
          "remote_url":"https://resc.deskline.net/images
/RPT/1/42a7bea5-9370-4624-b992-9d8cae21bd85/99/image.jpg"
        }
      ],
      "startDate": "2021-11-09T19:00:00.000+0000",
      "endDate": "2021-11-09T22:59:59.000+0000",
    }
  }
]
```


Fig. 18.5 JSON data about an event from an IT provider

et al. 2020). Therefore, many applications of them use some sort of external knowledge or human interaction.

18.2.4 Handling of Dynamic and Active Data

The approaches we have seen so far are typically useful for the creation of various sizes but rather static knowledge. We can consider three kinds of data:

```
prefixes:
 schema: "https://schema.org/"
 myfunc: "http://myfunc.com/"

mappings:
 event:
  sources:
   - ['input~jsonpath', '$.*.event']
  s: http://example.org/$(external_id)
  po:
   - [a, schema:Event]
   - [schema:identifier, $(external_id)]
   - [schema:name, $(name_translations.de), de~lang]
   - [schema:description, $(description_translations.de), de~lang]
   - [schema:url, $(url_translations.de)]
   - [schema:startDate, {fn: myfunc:asDateTime, pms: [$(startDate)]}]
   - [schema:endDate, {fn: myfunc:asDateTime, pms: [$(endDate)]}]
   -[schema:image, {mapping: image, join: [external_id, ^^external_id]}]

 image:
  sources:
   - ['input~jsonpath', '$.*.event.images.*']
  po:
   - [a, schema:ImageObject]
   - [schema:url, $(remote_url)]
```

Fig. 18.6 The RML mapping for the data in Fig. 18.5

- *Static*: data have high stability and slow velocity (names, addresses, the height of a mountain, etc.)
- *Dynamic*: data that changes very frequently (e.g., weather forecast, stock prices)
- *Active*: data that can change the status of a resource (e.g., booking a hotel room)

The latter two types of data can be handled with Semantic Web services. We create semantic annotations of Web services, particularly with a focus on the following aspects:

- The input and output of a Web service and their relationship (functionality)
- The data being exchanged (information model)
- The order of operations (behavior), and
- Nonfunctional properties such as response time, price, provider, etc.

The initial efforts were more focused on Simple Object Access Protocol (SOAP) Web services. The Web Service Modeling Framework (WSMF) (Fensel and Bussler 2002) and Web Ontology Language for Services (OWL-S) (Martin et al. 2007) are

```
@prefix ns0: <https://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/15dc0b05-01d8-4dbe-9cf0-3f579c1fd3d7>
 a <https://schema.org/Event> ;
 ns0:description """Hurra, ab Montag ist wieder Wochenende! heißt das
neue Programm von Bernd Stelter. Ein Programm für alle, die über Mon-
tage mosern, über Dienstage diskutieren, die Mittwoche mies und Don-
nerstage doof finden. \n„Und was machst du am Wochenende?“ – „Party!“
Es geht um Work-Life-Balance. ... Tickets online buchen"""@de ;
 ns0:endDate "2021-11-09T22:59:59.000Z"^^xsd:dateTime ;
 ns0:identifier "15dc0b05-01d8-4dbe-9cf0-3f579c1fd3d7" ;
 ns0:image [
  a ns0:ImageObject ;
  ns0:url  "https://resc.deskline.net/images/RPT/1/42a7bea5-9370-
4624-b992-9d8cae21bd85/99/image.jpg"
 ] ;
 ns0:name "Bernd Stelter"@de ;
 ns0:startDate "2021-11-09T19:00:00.000Z"^^xsd:dateTime ;
 ns0:url "http://www.ahrtal.de" .
```

Fig. 18.7 RDF data created based on JSON data and RML mapping

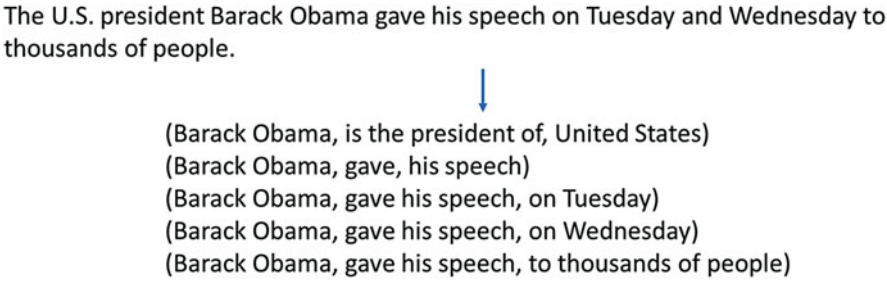


Fig. 18.8 OpenIE extracting triples without supervision from a sentence

two major examples of initial approaches to Semantic Web Services. More recently, lightweight approaches that target Web APIs[25] gained popularity. SPARQL-Microservices (Michel et al. 2018) is an approach that enables SPARQL queries over Web APIs to integrate them with Linked Data. Hydra (Lanthaler and Gütl 2013) provides an RDFS-based vocabulary to apply Linked Data principles to the

[25] We use Web API as an umbrella term for Web services over HTTP that implement REST principles to some extent.

resources comprising Web APIs. Hydra Web APIs attach machine-readable affordances to resources that allow clients to consume the Web service in a hypermedia-driven way.

WASA[26] is an approach for annotating Web services based on schema.org and SHACL (Şimşek et al. 2018). Web services are annotated as a set of potential actions that can be taken on resources. Each action is an instance of the *schema:Action* type. A functionality is defined by a subtype of the action (e.g., Search, Add, Delete) and its input and output:

- Input and output are defined with domain specification operators in SHACL.
- Clients can automatically build requests that fit the domain-specific pattern defined by the potential action.

Behavior is described with potential action links defined on the responses, e.g., a room offer from a hotel returned by a Web service call can have a reserve action as a potential action.

18.2.5 Summary

Building knowledge graphs implies a structured lifecycle of activities. Here, we discussed the knowledge creation process that generates semantically annotated data from heterogeneous sources.

We often use schema.org as the golden standard and focus on generating assertion knowledge. The domain specification process is introduced in the knowledge creation methodology to migrate this golden standard to a domain and task-specific model as schema.org is too broad, coming with a very large size and irrelevant aspects, as well as too shallow, i.e., it does not provide deep enough coverage of specific domains.

Knowledge can be created manually, semi-automatically from content, semi-automatically from data, and automatically (to a certain extent). Actually, the majority of knowledge is created via mappings from data sources. Declarative mappings improve the reusability and portability of these mappings. Dynamic and active data are typically handled with Web services, which can semantically be annotated via various approaches.

References

Arenas-Guerrero J, Chaves-Fraga D, Toledo J, Perez MS, Corcho O (2022) Morph-KGC: scalable knowledge graph materialization with mapping partitions. Semantic Web (Preprint) 1–20

[26] http://wasa.cc

Benjamins R, Fensel D, Decker S, Gomez Perez A (1999) (KA)2: building ontologies for the internet: a mid-term report. Int J Hum Comput Stud 51(3):687–712
Bernaras A, Laresgoiti I, Corera J (1996) Building and reusing ontologies for electrical network applications. Proc ECAI 96(1996):298–302
Calvanese D, Cogrel B, Komla-Ebri S, Kontchakov R, Dv L, Rezk M, Rodriguez-Muro M, Xiao G (2017) Ontop: answering SPARQL queries over relational databases. Semantic Web 8(3): 471–487
Clark A, Fox C, Lappin S (eds) (2012) The handbook of computational linguistics and natural language processing, vol 118. Wiley
De Nicola A, Missikoff M, Navigli R (2005) A proposal for a unified process for ontology building: upon. In: Database and expert systems applications: 16th international conference, DEXA 2005, Copenhagen, Denmark, August 22–26, 2005. Proceedings 16, Springer, pp 655–664
Deng C, Ji X, Rainey C, Zhang J, Lu W (2020) Integrating machine learning with human knowledge. iScience 23(11):101656
Dimou A, Vander Sande M, Colpaert P, Verborgh R, Mannens E, Van de Walle R (2014) RML: a generic language for integrated RDF mappings of heterogeneous data. In: LDOW 1184
Fensel D, Bussler C (2002) The web service modeling framework WSMF. Electron Commer Res Appl 1(2):113–137
Fensel D, Erdmann M, Studer R (1997) Ontology groups: Semantically enriched subnets of the www. In: Proceedings of the 1st International Workshop Intelligent Information Integration during the 21st German Annual Conference on Artificial Intelligence, Freiburg, Germany, September
Fensel D, Simsek U, Angele K, Huaman E, Karle E, Panasiuk O, Toma I, Umbrich J, Wahler A (2020) Knowledge graphs. Springer
Fernández-López M, Gómez-Pérez A, Juristo N (1997) METHONTOLOGY: from ontological art towards ontological engineering. AAAI Conference on Artificial Intelligence
Garcia-Gonzalez H, Boneva I, Staworko S, Labra-Gayo JE, Lovelle JMC (2020) ShExML: Improving the usability of heterogeneous data mapping languages for first-time users. PeerJ Comput Sci 6:318
Goldbeck G, Ghedini E, Hashibon A, Schmitz G, Friis J (2019) A reference language and ontology for materials modelling and interoperability. https://publica.fraunhofer.de/handle/publica/406693
Gomez-Perez A, Fernandez-Lopez M, Corcho O (2006) Ontological engineering: with examples from the areas of knowledge management, e-Commerce and the Semantic Web. Springer
Gruber TR (1993) Toward principles for the design of ontologies used for knowledge sharing, knowledge systems laboratory. Computer Science Department, Stanford University, Stanford, CA
Gruninger M, Fox MS (1995) Methodology for the design and evaluation of ontologies. In: Proceedings of IJCAI'95, Workshop on Basic Ontological Issues in Knowledge Sharing
Guha RV (1991) Contexts: a formalization and some applications. Stanford University
Hepp M (2005) eClassOWL: a fully-fledged products and services ontology in OWL. In: The Poster Proceedings of International Semantic Web Conference (ISWC) 2005, Galway, Ireland
Heyvaert P, De Meester B, Dimou A, Verborgh R (2018) Declarative Rules for Linked Data Generation at your Finger-tips! In: Proceedings of the 15th ESWC: Posters and Demo
Iglesias E, Jozashoori S, Chaves-Fraga D, Collarana D, Vidal ME (2020) SDM-RDFizer: An RML interpreter for the efficient creation of RDF knowledge graphs. In: Proceedings of the 29th ACM International Conference on Information and Knowledge Management, Association for Computing Machinery, New York, NY, USA, CIKM'20, pp 3039–3046. https://doi.org/10.1145/3340531.3412881
Kärle E, Simsek U, Fensel D (2017) semantify.it, a platform for creation, publication and distribution of semantic annotations. In: Proceedings of SEMAPRO 2017: The Eleventh International Conference on Advances in Semantic Processing, Barcelona, November 25–29, pp 22–30

Kotis K, Vouros GA, Alonso JP (2005) HCOME: a tool-supported methodology for engineering living ontologies. In: Semantic Web and Databases: Second International Workshop, SWDB 2004, Toronto, Canada, August 29–30, 2004, Revised Selected Papers 2, Springer, pp 155–166

Lanthaler M, Gütl C (2013) Hydra: a vocabulary for hypermedia-driven Web APIs. LDOW 996: 35–38

Lefrançois M, Zimmermann A, Bakerally N (2017) Flexible RDF generation from RDF and heterogeneous data sources with SPARQL-Generate. In: Knowledge Engineering and Knowledge Management: EKAW 2016 Satellite Events, EKM and Drift-an-LOD, Bologna, Italy, November 19–23, 2016, Revised Selected Papers, Springer, pp 131–135

Lenat DB, Guha RV (1989) Building large knowledge-based systems; representation and inference in the Cyc project. Addison-Wesley

Li H, Armiento R, Lambrix P (2020) An ontology for the materials design domain. In: Pan JZ, Tamma V, d'Amato C, Janowicz K, Fu B, Polleres A, Seneviratne O, Kagal L (eds) The Semantic Web – ISWC 2020. Springer, Cham, pp 212–227

Lourdusamy R, John A (2018) A review on metrics for ontology evaluation. In: 2018 2nd International Conference on Inventive Systems and Control (ICISC), pp 1415–1421. https://doi.org/10.1109/ICISC.2018.8399041

Manning CD, Surdeanu M, Bauer J, Finkel JR, Bethard S, McClosky D (2014) The Stanford CoreNLP natural language processing toolkit. In: Proceedings of 52nd annual meeting of the association for computational linguistics: System demonstrations, pp 55–60

Martin D, Burstein M, McDermott D, McIlraith S, Paolucci M, Sycara K, McGuinness DL, Sirin E, Srinivasan N (2007) Bringing semantics to web services with OWL-S. World Wide Web 10: 243–277

Masolo C, Borgo S, Gangemi A, Guarino N, Oltramari A (2003) WonderWeb deliverable D18: ontology library. Laboratory for Applied Ontology, ISTC-CNR

Mausam M (2016) Open information extraction systems and downstream applications. In: Proceedings of the twenty-fifth international joint conference on artificial intelligence, pp 4074–4077

Maynard D, Bontcheva K, Augenstein I (2016) Natural language processing for the semantic web. In: Ding Y, Groth P (eds) Synthesis lectures on the semantic web: theory and technology, vol 15. Morgan & Claypool Publishers, pp 1–184

Michel F, Djimenou L, Zucker CF, Montagnat J (2017) xR2RML: relational and non-relational databases to RDF mapping language. Technical report, CNRS

Michel F, Faron-Zucker C, Gandon F (2018) Bridging Web APIs and linked data with SPARQL micro-services. In: The Semantic Web: ESWC 2018 Satellite Events: ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3–7, 2018, Revised Selected Papers 15, Springer, pp 187–191

Mitchell T, Cohen W, Hruschka E, Talukdar P, Yang B, Betteridge J, Carlson A, Dalvi B, Gardner M, Kisiel B et al (2018) Never-ending learning. Commun ACM 61(5):103–115

Neches R, Fikes RE, Finin T, Gruber T, Patil R, Senator T, Swartout WR (1991) Enabling technology for knowledge sharing. AI Mag 12(3):36–36

Noy NF, McGuinness DL (2001) Ontology development 101: a guide to creating your first ontology. Stanford University, Stanford, CA

Paulheim H (2018) Machine learning with and for semantic web knowledge graphs. In: Reasoning web learning, uncertainty, streaming, and scalability: 14th International Summer School 2018, Esch-sur-Alzette, Luxembourg, September 22–26, 2018, Tutorial Lectures 14, pp 110–114

Pinto HS, Staab S, Tempich C (2004) Diligent: towards a fine-grained methodology for distributed, loosely-controlled and evolving engineering of ontologies. ECAI 16:393

Poveda-Villalon M, Fernandez-Izquierdo A, Fernandez-Lopez M, Garcia-Castro R (2022) LOT: an industrial oriented ontology engineering framework. Eng Appl Artif Intell 111:104755. https://doi.org/10.1016/j.engappai.2022.104755. https://www.sciencedirect.com/science/article/pii/S0952197622000525

Radford A, Narasimhan K, Salimans T, Sutskever I, et al (2018) Improving language understanding by generative pre-training. Technical report. https://www.cs.ubc.ca/amuham01/LING530/papers/radford2018improving.pdf

Sack H (2015) Ontology alignment. OpenHPI tutorial - knowledge engineering with semantic web technologies. https://open.hpi.de/courses/semanticweb2015/items/2d5hp8qEjA2Mm0rXLtLG4v

Schreiber AT, Schreiber G, Akkermans H, Anjewierden A, Shadbolt N, de Hoog R, Van de Velde W, Wielinga B (2000) Knowledge engineering and management: the CommonKADS methodology. MIT Press

Şimşek U, Kärle E, Fensel D (2018) Machine readable Web APIs with schema.org action annotations. Procedia Comput Sci 137:255–261

Şimşek U, Kärle E, Fensel D (2019) RocketRML - a NodeJS implementation of a use-case specific RML mapper. In: Proceedings of 1st Knowledge Graph Building Workshop co-located with 16th Extended Semantic Web Conference (ESWC), Portoroz, Slovenia, June 3, 2019. CEUR, vol 2489

Şimşek U, Angele K, Kärle E, Panasiuk O, Fensel D (2020) Domain-specific customization of schema.org based on SHACL. In: The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II 19, Springer, pp 585–600

Simsek U, Kärle E, Angele K, Huaman E, Opdenplatz J, Sommer D, Umbrich J, Fensel D (2022) A knowledge graph perspective on knowledge engineering. SN Comput Sci 4(1):16

Studer R, Benjamins VR, Fensel D (1998) Knowledge engineering: Principles and methods. Data Knowl Eng 25(1–2):161–197

Suarez-Figueroa MC (2007) D5.3.1 NeOn development process and ontology life cycle. NEON Project Consortium, Technical Report

Sure Y, Staab S, Studer R (2004) On-to-knowledge methodology (OTKM). In: Handbook on ontologies. Springer, pp 117–132

Swartout B, Patil R, Knight K, Russ T (1996) Toward distributed use of large-scale ontologies. In: Proceedings of the tenth workshop on knowledge acquisition for knowledge-based systems, vol 138, p 25

Uschold M, King M (1995) Towards a methodology for building ontologies. Technical report

Chapter 19
Knowledge Hosting



Conceptually, knowledge graphs are a set of nodes and edges between them that represent entities and their relationships. Different storage paradigms and implementations can be used to host a knowledge graph. This chapter focuses on these different paradigms and explains their advantages and disadvantages. Each paradigm will be demonstrated with a small example from the German Tourism Knowledge Graph (Fig. 19.1).[1] The German Tourism Knowledge Graph (GTKG) integrates tourism-related data from 16 regional marketing organizations in Germany. It contains a total of ~60K instances of accommodation providers, events, points of interest (POIs), and touristic tours.[2] The example we extracted from GTKG contains a *schema:Hotel* instance (*dzt-entity:166417787*) that is described with the *schema:address*, *schema:description*, *schema:geo*, and *schema:name* properties. Moreover, the type of *schema:Hotel* is defined as a subclass of *schema:LocalBusiness*.

In the remainder of this chapter, we first introduce the hosting-related challenges for knowledge graphs. Then we introduce different paradigms that can be used to host Resource Description Framework (RDF) graphs and illustrate them with small examples. Then we explain RDF triplestores in more extensive detail. Finally, we provide a larger illustrative example based on the German Tourism Knowledge Graph and conclusions.

[1] https://open-data-germany.org/datenbestand/ and https://open-data-germany.org/datenbestand-such-widget/

[2] Status in July 2023.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_19

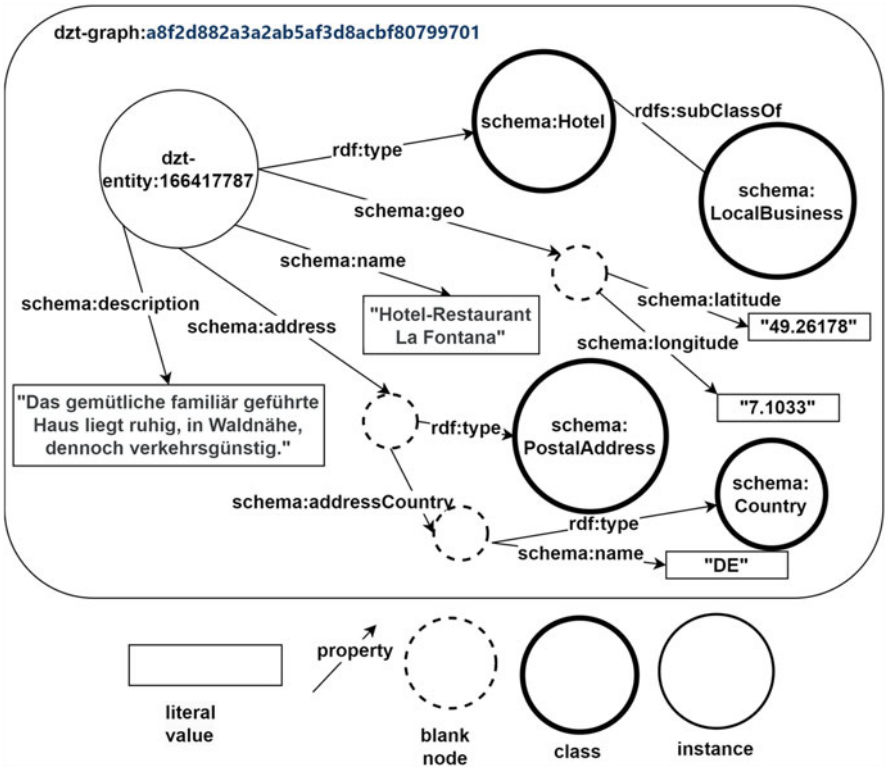


Fig. 19.1 An excerpt from the German Tourism Knowledge Graph

19.1 Challenges in Hosting Knowledge Graphs

Hosting a knowledge graph has the following challenges related to size, data model, heterogeneity, velocity, different points of view, and deployment. In the following, we will briefly explain these challenges:

- *Size*: One characteristic of knowledge graphs is their vast size. A knowledge graph might contain billions of facts. Storing, hosting, maintaining, and deploying such a vast number of facts are quite challenging.
- *Data model*: Another challenge may occur due to the data model of a knowledge graph. A knowledge graph is technically a semantic network. A semantic network is a directed or undirected graph consisting of vertices, which represent concepts, and edges, which represent semantic relations between concepts.[3] The challenge is to find the most suitable way to represent data in graph form.

[3] See https://en.wikipedia.org/wiki/Semantic_network and Part I.

- *Heterogeneity*: Knowledge graphs by nature have a flexible schema. The hosting solution should be able to flexibly support the integration of data with heterogeneous schemas.
- *Velocity*: Knowledge graphs change rapidly, e.g., new knowledge is inferred through reasoning based on the given facts, and new sources may get integrated. The hosting solution must be able to handle high-velocity changes.
- *Points of view*: Different use cases require different constraints applied to the data stored in a knowledge graph. The hosting solution should ideally provide a mechanism to contextualize the data to support potentially conflicting constraints and inference rules for different use cases.
- *Deployment*: Knowledge graphs need to be accessible for different kinds of applications. Ideally, the hosting solution provides various Application Programming Interfaces (APIs) and query interfaces and supports different formats to enable a variety of applications.

In the following, we will analyze different hosting paradigms and their pros and cons with regard to the challenges mentioned above.

19.2 Knowledge Hosting Paradigms

Although knowledge graphs have a graph data model logically, they can be hosted in databases with different hosting paradigms. In this section, we will introduce the three most popular hosting paradigms for knowledge graphs, namely, relational databases, document stores, and graph databases. We will present each paradigm to a certain extent and explain their advantages and disadvantages in the scope of the challenges in hosting knowledge graphs.

19.2.1 Relational Databases

The *relational model* (Codd 1970) decouples logical representation from physical implementation. In other words, it separates data from hardware and the application logic (program). Data are stored in relations. The operations are conducted at the relational level rather than at the tuple level, which means they can be done on many tuples at once. The operations on relations are formalized by relational algebra, which is based on set operations. This algebra provides applications with an abstract access layer to access, store, and modify data. Relational algebra is reflected in a high-level declarative language, called *Structured Query Language (SQL)*,[4] to query relational databases.

[4] https://en.wikipedia.org/wiki/SQL

Table 19.1 An example table in a relational database

Customer ID	Tax ID	Name	Address	[More fields...]
1234567890	555-5512222	Ramesh	323 Southern Avenue	...
2223344556	555-5523232	Adam	1200 Main Street	...
3334445563	555-5533323	Shweta	871 Rani JhansiRoad	...
4232342432	555-5325523	Sarfaraz	123 Maulana Azad Sarani	...

Table 19.2 Join operation on *Employee* and *Dept* tables and their result

a. *Employee* table

Name	EmpID	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

b. *Dept* table

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

c. Result of *Employee* ⋈ *Dept*

Name	EmpID	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

The relational model stores data in tuples in a structure called relations (tables). A relation consists of a header (a finite set of attribute names (columns)) and a body (a set of tuples). For example, Customer (*Customer_ID*, Tax_ID, Name, Address, City, State, Zip, Phone, Email, Sex) represents a relation and its columns. In this relation, Customer_ID is the primary key that uniquely identifies a tuple (row)). Table 19.1 shows an example table.[5]

There are various operations that can be done on a relation. These operations include but are not limited to *Join*, *Projection*, and *Selection* operations.

The *Join* operation combines the tuples from two tables (cartesian product) and eliminates the ones that do not fit the join condition. For example, natural join (⋈) only retrieves the tuples from the combined table with the same value on their common fields.[6] The example in Table 19.2[7] shows how the *Join* operation works for the relations *Dept* and *Employee*. Tables 19.2a and 19.2b are joined on the

[5] Taken from https://en.wikipedia.org/wiki/Relational_model

[6] See https://en.wikipedia.org/wiki/Relational_algebra for different kinds of join operations. A join operation without any join condition produces a cartesian product.

[7] Examples taken from https://en.wikipedia.org/wiki/Relational_algebra#Natural_join_(⋈)

Table 19.3 Projection operation

a. *Person* table

Name	Age	Height
Harry	34	180
Sally	28	164
George	28	170
Helena	54	154
Peter	34	180

b. Projection of the table *Person*: $\Pi_{Age,\ Height}$ (*Person*)

Age	Height
34	180
28	164
28	170
54	154
34	180

Table 19.4 Selection operation

a. *Person* table

Name	Age	Weight
Harry	34	80
Sally	28	64
George	28	70
Helena	54	54
Peter	34	80

b. Selection of the tuples that have an *Age* column value equal to or greater than 34 from the table *Person*: $\sigma_{Age \geq 34}$ (Person)

Name	Age	Weight
Harry	34	80
Helena	54	54
Peter	34	80

DeptName columns in the *Employee* and *Dept* tables. The result is shown in Table 19.2c. Here, the row *Production* in the *Dept* table is eliminated as there is no common value for the *DeptName* column in the *Employee* table for that row.

The *Projection (𝕀𝕀)* operation[8] applies vertical filtering to a table. It retrieves only a subset of the attribute (column) values. The *Person* relation in Table 19.3a is projected to the *Age* and *Height* columns shown in Table 19.3b.

The *Selection (σ)* operation[9] applies horizontal filtering to a table. It retrieves only a subset of tuples (rows). Table 19.4 shows a selection example. The *Person* relation

[8] Example taken from https://en.wikipedia.org/wiki/Projection_(relational_algebra)

[9] Example taken from https://en.wikipedia.org/wiki/Selection_(relational_algebra)

Table 19.5 Statement table representation of a part of the knowledge in Fig. 19.1

Subject	Predicate	Object
Hotel	subClassOf	Thing
Hotel	subClassOf	LocalBusiness
166417787	type	Hotel
166417787	description	"Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig."
166417787	address	_:123
_:123	addressCountry	_:456
_:456	type	Country
_:456	name	"DE"
...	...	...

in Table 19.4a goes under a selection operation in Table 19.4b, where only the tuples whose *Age* column has a value equal to or greater than 34 are present.

Various other set operations, like set union and set difference, also exist; however, they are out of the scope of this book. We refer the readers to a plethora of textbooks about relational databases (e.g., Connolly and Begg 2005; Harrington 2016).

Storing a large knowledge graph in a relational database may result in large tables. Depending on the storing approach (e.g., statement table approach), a single table may contain all the knowledge graph facts (i.e., a single table containing three columns for the subject, predicate, and object); see Table 19.5. For querying such a knowledge graph, potentially vast amounts of data need to be joined via self-joins. In this case, the query would work on the cartesian product of the same table (i.e., a knowledge graph with 1M triples means we might potentially have to work with 10^{12} rows in the memory in a naïve implementation).

Another issue is domain and range definitions as defined by RDF Schema (RDFS). A domain defines the classes the properties apply to. A range defines the type of value the property can take. Although integrity constraints may be used to implement domain and range restrictions natively to some extent in a closed-world setting, representing domain and range definitions and their semantics for languages like RDF Schema must be handled by applications.

Similar issues are also encountered while representing the semantics for the class and property hierarchies. The representation of class and property hierarchies may require many auxiliary tables and joins. The application needs to hardwire the semantics. The inheritance of properties between subclasses is also problematic since the inheritance of ranges by subclasses must be handled by the database designer.

The bottom line is that for many reasons the application logic must hardwire the semantics of the modeling languages, like RDFS, which harms the declarative nature and reusability of knowledge graphs. In the following, we will cover four main approaches for hosting knowledge graphs in relational databases (Ben Mahria et al. 2021).

Table 19.6 Class-centric storage of a part of the knowledge in Fig. 19.1 in a relational database. The columns with NULL values are omitted for conciseness

a. *Class* table

ID	subClassOf
LocalBusiness	Thing
Hotel	LocalBusiness

b. *Hotel* table

ID	Type	Description	Address	Name	Geo
166417787	Hotel	"Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig."	_:123	"Hotel-Restaurant La Fontana"	_:789

c. *Country* table

ID	Name
_:456	"DE"

d. *PostalAddress* table

ID	addressCountry
_:123	_:456

A *statement table* is the most straightforward approach. The graph is stored in one table with three columns (subject, predicate, object). This results in one large table containing all the graph information. Table 19.5 shows the partial storage of the knowledge graph shown in Fig. 19.1.

Statement tables are a simple way to host a knowledge graph in a relational database. RDF triples representing the knowledge graph can directly be stored in this table without any change, but this approach has quite some drawbacks. First, the data are not normalized. Therefore, value replications can happen. Second, a growing number of triples result in inefficient self-joins (e.g., for traversing the hierarchy of classes).

The *class-centric table* approach uses one table for each type. All property values for a class are stored within a single table. The properties can appear in multiple tables if other classes use the same properties. Table 19.6 shows a partial class-centric representation of the knowledge graph shown in Fig. 19.1. Each class in the knowledge graph is represented as a table and its properties as columns. The properties of a type without any value assertion in the knowledge graph may need to be created as columns with NULL values. In this case, the tables could be very sparse (with many null values). In addition to the class tables, we have a table to host the classes and their hierarchy (Table 19.6a), which helps answer queries like "give me all *LocalBusiness* instances" as this requires a join between the class table (Table 19.6a) and *Hotel* table (Table 19.6b).

For each type, a separate table is used to store its properties, so it is more intuitive from a modeling point of view. Still, this approach has some drawbacks. First, adding new properties and classes is cumbersome as the schema must be recompiled. For example, if we define an instance of a type *Event*, first, a table and columns for the *Event* type and its properties must be created in the schema of the relational

Table 19.7 Property-centric storage of a part of the knowledge in Fig. 19.1 in a relational database

Type		**Geo**	
Subject	**Object**	**Subject**	**Object**
166417787	Hotel	166417787	_:789
Name		**Address**	
Subject	**Object**	**Subject**	**Object**
166417787	"Hotel-Restaurant La Fontana"	166417787	_:123
Description			
Subject	**Object**		
166417787	"Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig"		

database. Second, the level of normalization is not enough to handle multivalued properties as they lead to repetitions of tuples for each value of a property. For example, if a hotel has multiple descriptions, we would need to create another tuple for the same hotel with a different description but repetitive values for all other columns.

Property-centric tables use one table per property**.** Each table contains two columns, namely, subject and object. This approach easily allows multivalued properties, although duplication of subjects is still necessary. Table 19.7 shows the partial representation of the property-centric storage of the knowledge graph in Fig. 19.1. For the sake of conciseness, only the properties defined for the *Hotel* instance are shown.

This approach also has similar drawbacks as the class-centric approach, however from a property-oriented perspective. First, adding a new property requires the creation of a new table and, therefore, a recompilation of the schema. A common operation like retrieving properties defined on a single instance requires joins over a vast number of tables.

Finally, *virtual RDF graphs* over relational databases do not implement a concrete knowledge graph with a relational database but provide an ontology-based access layer on top of existing relational databases. In this case, SPARQL queries are transformed into SQL queries and executed on the relational database. Results are then transformed back into solution mappings for SPARQL.

Virtual RDF graphs have been a popular way to convert relational databases to knowledge graphs in recent years. Figure 19.2 shows the process model of querying a virtual RDF graph. The SPARQL query q is rewritten with respect to the ontology. The middleware unfolds the query q with respect to the mapping from the schema of the data source to the ontology. The generated SQL query is ready to be evaluated over the data source. Taking again the mappings into account, values in the SQL answers are used to build RDF terms (Xiao et al. 2019).

A direct implementation of this workflow may be highly inefficient. A number of optimizations are required (Xiao et al. 2019), e.g.:

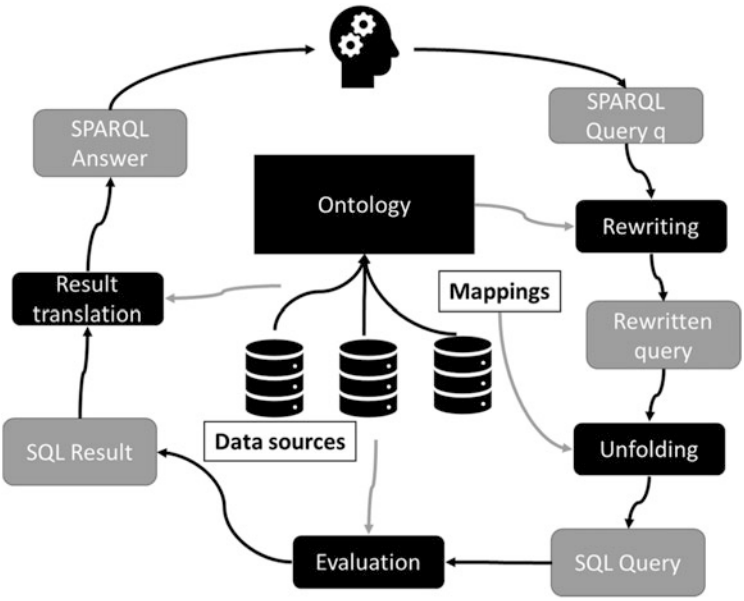


Fig. 19.2 The process model of querying a virtual RDF graph, adapted from Xiao et al. (2019)

- Compiling the ontology into the mapping in an offline phase
- Exploiting the constraints over the data to strongly simplify the queries after the unfolding phase, and
- Planning query execution using a cost-based model

An example implementation of such an approach is Ontop[10] (Xiao et al. 2020). It is a virtual RDF graph framework distributed with the Apache 2 license.[11] It supports a customized mapping language as well as RDB to RDF Mapping Language (R2RML).[12] It supports a subset of SPARQL 1.1 and contains many optimizations for *Join*, *Union*, and LeftJoin operations. It implements reasoning via query rewriting, which supports the OWL 2 QL profile.

In general, virtual RDF graphs have the following advantages:

- They do not require any preprocessing on a relational database management system (RDBMS).
- They allow ontology-based access via mappings and query rewriting (e.g., SPARQL to SQL).

[10] https://ontop-vkg.org

[11] Meanwhile, also available as a commercial product under the name of Ontopic.

[12] https://www.w3.org/TR/r2rml/

- They provide a relatively cheap way to build a knowledge graph from relational databases.
- They provide a smooth integration in industrial standard software environments.

However, they also have the following drawbacks:

- Many things can go wrong with query rewriting and unfolding (mappings need extra attention).
- Querying the schema is challenging (due to the underlying relational model).
- Typically, only limited querying and reasoning capabilities are provided.

To summarize, although relational databases can be used to directly host knowledge graphs, they still suffer from some issues.[13] Revisiting the challenges presented in Sect. 19.1, we can particularly see drawbacks regarding data model, heterogeneity, and velocity. Representing graph models in relational databases typically requires many queries and expensive joins. The relational model is not particularly suitable for representing knowledge graphs as they typically adopt a flexible schema, so much so that the line between data and schema typically disappears. Relational databases are not suitable for such a scenario as the schema is rigid and strictly decoupled. Also, the velocity challenge is a barrier as any inference resulting in new TBox knowledge may require the recompilation of the schema, which can be an expensive operation. Moreover, representing relationships like inheritance can be tricky, and implementing entailment rules even for RDFS is not entirely possible in a straightforward way without getting the application logic or externally defined rules involved.

19.2.2 Document Model

The document model stores data as nested key-value pairs,[14] and in many implementations, they are organized in *documents* (*hence the name*), *akin to tuples in relational databases.* Document stores have become popular with the rise of big and streaming data as they have no rigid schema and can be scaled up rather easily. Documents are organized in so-called *Collections* in many implementations. Collections are analogous to tables in relational databases, but they do not enforce a schema; therefore, each document stored in a collection can have a different structure and metadata.

A document model can be used for hosting knowledge graphs in two ways, namely, as nested objects or with the adoption of document references. The *nested object* approach is the native way of storing documents where everything about an instance is stored in one document. Although this makes it easier to access individual

[13] See also here for an interesting analysis. https://doriantaylor.com/there-is-no-sqlite-for-rdf

[14] Therefore, they are also sometimes referred to as a special kind of key-value pair databases. https://en.wikipedia.org/wiki/Document-oriented_database#Relationship_to_key-value_stores

```
{
        "_id": "166417787",
        "@type": "Hotel",
        "description": "Das gemütliche familiär geführte Haus liegt ruhig,
in Waldnähe, dennoch verkehrsgünstig.",
        "geo": {
                "latitude": "49.26178",
                "longitude": "7.1033",
        },
        "name": " Hotel-Restaurant La Fontana ",
        "address": {
                "addressCountry": {
                        "@type": "Country",
                        "name": "DE"
                }
    }
}
```

Fig. 19.3 An indicative nested object representation of the knowledge graph in Fig. 19.1

documents, the updates are problematic as a nested object appearing in many documents needs to be updated, and this is typically handled by application logic or middleware. Figure 19.3 represents the knowledge graph in Fig. 19.1 as nested objects. Each nested object without an *id* field represents a blank node.

The *document reference* approach splits the documents into conceptually cohesive collections. In this scenario, instead of nesting objects, references to them are stored. This eliminates the disadvantage of nested objects when it comes to updates; however, in return, it requires the implementation of *Join* operations by the application and blurs the difference between document stores and relational databases. Figure 19.4 shows a partial representation of the knowledge graph in Fig. 19.1. One collection is used for the *Hotel* instances; another is for *PostalAddress* instances. The value of the *address* property of a *Hotel* instance is a blank node in the knowledge graph. Therefore, in the document store, the instance in the *PostalAddress* collection has an internally unique ID (i.e., *_:123*), and this ID is used as the value for the address property on the Hotel document.

There are many document store implementations[15] that can be used to host knowledge graphs. For example, HexaDB[16] implements an RDF triplestore with a key-value-based document database backend called RocksDB.[17] It implements certain optimizations via indexing. AllegroGraph[18] also provides an alternative document store backend alongside a graph-based model.

[15] A list can be found here: https://en.wikipedia.org/wiki/Category:Document-oriented_databases

[16] https://github.com/angshuman/hexadb

[17] An open source project from Meta http://rocksdb.org/

[18] https://allegrograph.com

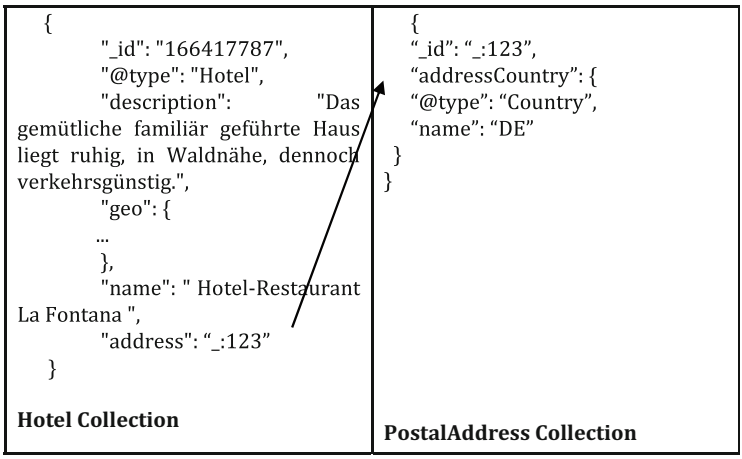


Fig. 19.4 Document reference representation of the knowledge graph from Fig. 19.1

The document or key-value model can be used to store knowledge graphs and provides some features that are desirable, such as being schemaless and therefore having looser consistency checking strategies. This helps with heterogeneity and velocity. Instances can be stored with different metadata, and multiple values for a property can be stored more naturally, which eliminates the disadvantage of a relational database that may lead to null values or duplications. Moreover, native support provided to JavaScript Object Notation (JSON) documents by many implementations provides an O(1) access to retrieve instances stored as documents in JavaScript Object Notation for Linked Data (JSON-LD) format. In this case, an instance is represented as a document.

The document store paradigm also emphasizes an important distinction between hosting knowledge for semantic annotation and that for building a knowledge graph. In the first scenario, the main deployment goal is to publish machine-understandable annotations of the content of Web pages. The most efficient way to do this is to create a one-to-one connection between the semantic annotation (e.g., a JSON-LD document) and the Web page it belongs to. This is relatively straightforward and efficient with document stores as each document corresponds to one annotation. In the latter scenario, document stores are not the best solution because it is harder to benefit from the connectedness of the graph data and support reasoning. Facts can be grouped together and efficiently queried or used for reasoning instead of browsing numerous documents that may contain them. We will discuss this in the next section and in Sect. 23.1.6 in more detail.

19.2.3 Graph Databases

The graph database paradigm represents data and/or the schema as graphs. Data manipulation is expressed by graph transformations. Many graph database implementations support a flexible schema. However, some provide data integrity features, like constraints, identity, and referential integrity (Angles and Gutierrez 2008). The graph data model consists of *nodes* representing the entities and *edges* representing the relationships between entities. Nodes and edges may have different levels of metadata attached to them, which distinguish graph database paradigms in terms of their capabilities. In the context of knowledge graphs, we focus on two popular types of graph databases: *property graphs* and *RDF triplestores*. We will cover property graphs briefly here, while we cover RDF triplestores in the following section separately.

Property graphs are graph models that allow the description of predicates connecting two entities. *Nodes* represent entities in the graph, and each node can hold any number of property assertions represented as key-value pairs. Nodes are taggable with labels representing the role in the graph, for example, the type of an instance a node is representing.

Edges represent a connection between two nodes and specify the meaning of the relationship between two nodes. An edge always contains a direction, a start and end node, and a label that indicates the type of relationship. Additionally, a relationship can also have property value assertions.

There is currently no standard for property graphs, and their development is mainly driven by industry. A widely adopted property graph implementation is Neo4J.[19] The graphs hosted in Neo4J can be queried with the Cypher[20] language.

An example query returning all hotels with the name Hotel-Restaurant La Fontana may be as follows:

```
MATCH (h:Hotel {name: 'Hotel-Restaurant La Fontana'})
RETURN h
```

Graph databases address the challenges of hosting knowledge graphs quite extensively. Many graph database implementations support hosting with flexible schemas to support heterogeneity and velocity. Obviously, the data model is also natively supported.

Property graphs are one of the most widespread implementations of graph databases for knowledge graphs. They allow the representation of entities with an arbitrary number of property value assertions and their relationships, which also can be described with properties. The main disadvantage of property graphs for knowledge graphs is the lack of standardized technologies for querying as well as the lack

[19] https://neo4j.com/product/neo4j-graph-database/

[20] https://neo4j.com/developer/cypher/querying/#cypher-examples

of native reasoning capabilities. However, there are some attempts to bridge these gaps between property graphs and RDF, which we will discuss in Sect. 19.3.

19.3 RDF Triplestores

RDF graph databases are optimized for the triple structure of RDF and, therefore, are often called triplestores. They store statements in the form of <*subject*> <*predicate*> <*object*> following the RDF data model.

Triplestores support reasoning based on RDFS and OWL2 and typically come with built-in reasoners. SPARQL 1.1 is supported as the query language, and many of them also provide APIs for retrieval and manipulation. Although triplestores may have different backend implementations, including relational databases and document stores (Karvinen et al. 2019), many recent implementations natively support the graph structure of RDF.

As with many other graph databases, triplestores provide storage with flexible schema; in fact, the difference between schema and data can disappear for the purposes of querying. Triplestore implementations are built based on solid standards from the World Wide Web Consortium (W3C), such as RDF, RDFS, Web Ontology Language (OWL), and SPARQL Protocol and RDF Query Language (SPARQL), which makes it easier to migrate between different vendors and develop tooling around them. There are many triplestore implementations that are used in industrial applications, and covering all of them here would require another book. We refer readers to a recent survey (Ali et al. 2021) covering 135 different triplestore implementations. It is known that they can scale up to trillions of triples for hosting knowledge graphs.[21]

The main difference between RDF graph databases and property graphs is the ability to make statements about relationships between two nodes and native reasoning capabilities. Property graphs natively support "making statements about statements" but do not have native reasoning support. On the other side of the coin, RDF databases have native reasoning support via their ontology languages, but due to the limitations of the RDF model, making statements about statements is cumbersome.

Still, with extensions like Named Graphs and RDF-Star, which we already covered in Sect. 13.1.1, additional metadata to a triple can be added to describe, for instance, the temporal and spatial validity of a statement, provenance, and other contextual information.

From both camps, there are efforts to bridge the gap between these two different graph models.[22] From the property-graph community, there are efforts to give

[21] https://www.w3.org/wiki/LargeTripleStores

[22] There was even a W3C workshop in 2019 to explore the possibilities for bridging the gap https://www.w3.org/Data/events/data-ws-2019/report.html

semantic capabilities to property graphs. For example, Neo4J supports a plugin called Neosemantics,[23] which enables the usage of Semantic Web Stack technologies, such as RDF(S), OWL, Simple Knowledge Organization System (SKOS), and SPARQL, with property graphs. Moreover, there are graph database implementations, like AnzoGraph, that support both property graph and RDF data models. Further standardization efforts for property graph[24] and general graph databases[25] are ongoing.

19.4 Illustration: German Tourism Knowledge Graph in GraphDB

The German Tourism Knowledge Graph is hosted in an RDF triplestore called GraphDB from Ontotext.[26] Like many other commercial triplestores, GraphDB provides all major functionalities needed for hosting knowledge graphs, such as:

- Storing
- Querying
- Visualization, and
- Reasoning of RDF graphs

In this section, we will demonstrate what these functionalities look like with the help of our running example presented in Fig. 19.1.

19.4.1 Storing an RDF Graph

The German Tourism Knowledge Graph is hosted in a GraphDB Enterprise instance and currently contains over 13M statements.[27] It integrates tourism-related data from 16 German states, such as instances of *events*, *tours*, and *points of interest (POIs).* The statements are organized in named graphs, where each named graph contains statements that are coming from the tourism marketing organization of a specific state. For example, the named graph identified with *dzt-graph:a8f2d882a3a2ab5af3d8acbf80799701* contains data from Saarland. This way, the provenance of each statement can be tracked, and features like licensing access control can be provided in an elegant way.

[23] https://neo4j.com/labs/neosemantics/
[24] https://opencypher.org
[25] https://www.gqlstandards.org/
[26] https://graphdb.ontotext.com/
[27] Status in December 2022.

subject	predicate	object
dzt-entity:-1664177870	onlim-schema:affectedBy	onlim-activity:997bedee-3ca2-4798-8d07-1c43dfe852bd
dzt-entity:-1664177870	http://purl.org/dc/terms/description	Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig. Der ideale Ausgangspunkt für den geschäftlichen oder privaten Aufenthalt.
dzt-entity:-1664177870	http://purl.org/dc/terms/title	Hotel-Restaurant La Fontana
dzt-entity:-1664177870	rdf:type	https://odta.io/voc/PointOfInterest

Fig. 19.5 A part of the graph in Fig. 19.1 browsed via the GraphDB user interface

GraphDB provides various possibilities to load an RDF graph. An RDF file can be uploaded via a graphical user interface (GUI), through an API, and via SPARQL INSERT/INSERT DATA queries.

Figure 19.5 shows an excerpt of the graph shown in Fig. 19.1 stored in GraphDB in tabular view. The subject of each statement is *dzt-entity:-1664177870*, and this subject is described with several properties. Although not visible in Figure 19.5, the statements are organized in the graph identified with the IRI *dzt-graph:a8f2d882a3a2ab5af3d8acbf80799701*.

The named graph is used to track the provenance of the statements. The IRI identifying a named graph can be used in the subject position of a triple to make further property value assertions about the named graph. For example, it is used to make property assertions about the dates on which the statements are added to the knowledge graph. Figure 19.6 shows the property assertions made on the named graph IRI. Terminology from schema.org and PROV-O[28] is used to describe the named graph. In this example, we see that the named graph represents a dataset, and triples were added to it on two different dates by two different processes, first in December 2021 and then in February 2022. Also, the IRI of the tourism marketing organization is attached via the *schema:publisher* property.

[28] https://www.w3.org/TR/prov-o/-PROV-O is a W3C standard OWL2 ontology to describe provenance information of arbitrary resources.

subject	predicate	object
dzt-graph:a8f2d882a	schema:publisher	onlim-entity:d0edf8f1-702e-4da8-98cf-9323112ae63c
dzt-graph:a8f2d882a	rdf:type	http://schema.org/Dataset
dzt-graph:a8f2d882a	rdf:type	prov:Entity
dzt-graph:a8f2d882a	prov:generatedAtTime	"2021-12-01T11:42:04"^^xsd:dateTime
dzt-graph:a8f2d882a	prov:generatedAtTime	"2022-02-28T15:31:21"^^xs

Fig. 19.6 Provenance information attached to the named graph (The named graph IRI is shortened for readability)

19.4.2 Querying

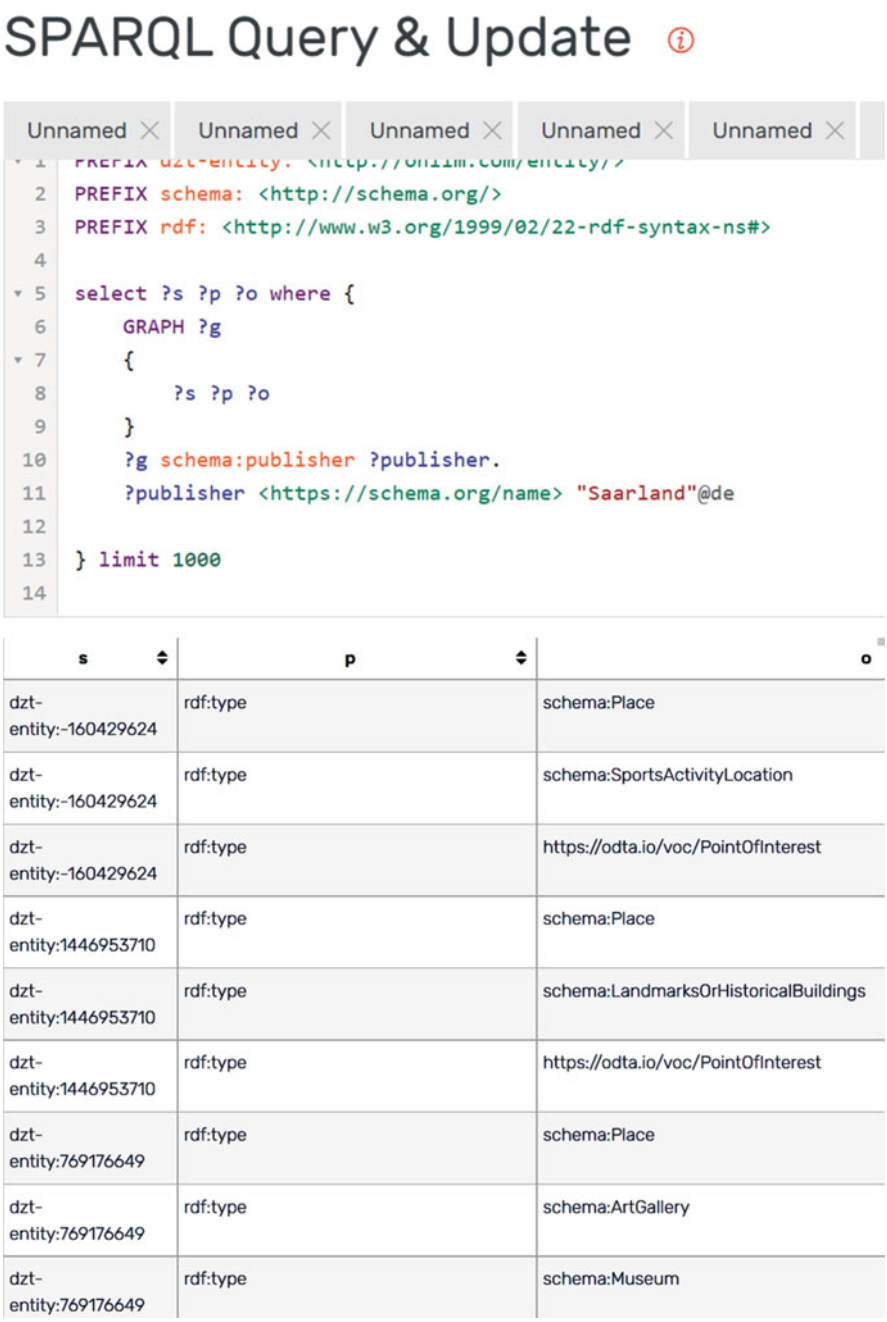
GraphDB provides a graphical interface and an API for running SPARQL 1.1 queries on a graph. Without specifying a named graph, all queries run on the default graph, which is the union of all named graphs in a repository. When a named graph is specified with the *FROM NAMED* or *GRAPH* keyword, then the scope of the triple patterns depends on the query.

Figure 19.7 shows the graphical interface that runs a query to retrieve 1000 triples from the named graph whose publisher's name is Saarland. This query requires the *GRAPH* keyword to bind the named graph IRI to a variable and then apply a filter on it based on the name property of the publisher.

19.4.3 Visualization

GraphDB provides various visualization possibilities for an RDF dataset. In the following, we will demonstrate three main visualizations, namely, the class hierarchy, the class relationships, and the visual graph.

The *class hierarchy* visualization provides a Venn-diagram-based view of the classes used in an RDF dataset. Each circle represents a class, and smaller circles inside a circle represent the subclasses of that class. The size of the circles is determined by the number of instances of a class in the dataset. For example,



s	p	o
dzt-entity:-160429624	rdf:type	schema:Place
dzt-entity:-160429624	rdf:type	schema:SportsActivityLocation
dzt-entity:-160429624	rdf:type	https://odta.io/voc/PointOfInterest
dzt-entity:1446953710	rdf:type	schema:Place
dzt-entity:1446953710	rdf:type	schema:LandmarksOrHistoricalBuildings
dzt-entity:1446953710	rdf:type	https://odta.io/voc/PointOfInterest
dzt-entity:769176649	rdf:type	schema:Place
dzt-entity:769176649	rdf:type	schema:ArtGallery
dzt-entity:769176649	rdf:type	schema:Museum

Fig. 19.7 GraphDB SPARQL interface with a query running on the German Tourism Knowledge Graph

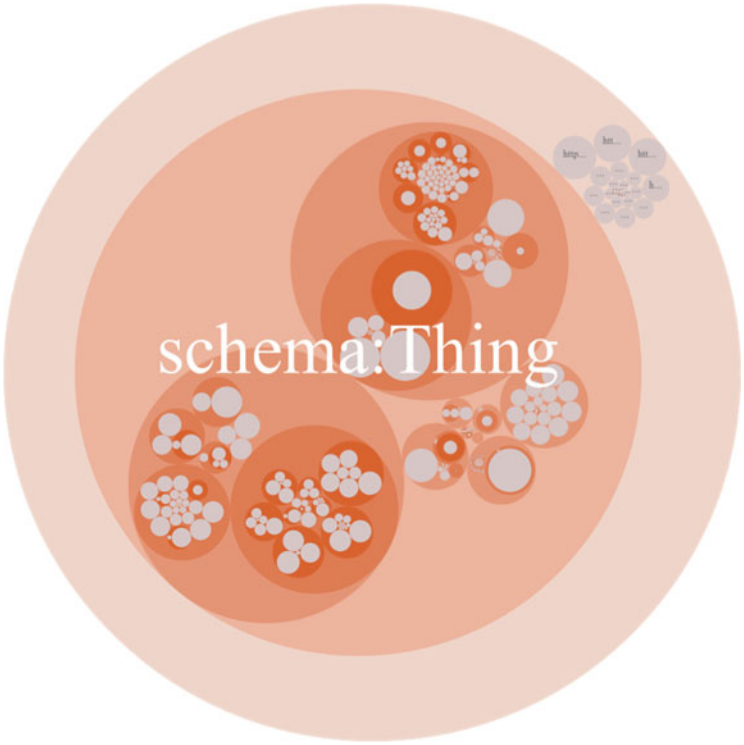


Fig. 19.8 Class hierarchy visualization of the German Tourism Knowledge Graph

Fig. 19.8 shows the classes in the German Tourism Knowledge Graph, where the biggest circle belongs to *schema:Thing*, which is one of the two highest classes in the hierarchies of schema.org.[29] This visualization is also a good way to do a "visual debugging" of the knowledge graph as it would make it visible if there are any "stray" circles that refer to some classes that occurred due to, e.g., a typo in the class IRI.

Class relationship visualization shows the number of links between the instances of pairs of classes. Figure 19.9 shows that there are ~520K links from *schema:Place* instances to *schema:GeoCoordinates* instances, and these links are provided with the *schema:geo* property. The large part of the circle between these two types indicates that they have significantly more links than the instances of the other types. This visualization can be used to see the most important types in the knowledge graph in terms of connectedness and usage.

Finally, the visual graph allows us to obtain a graph-based visualization given the IRI of a resource or a custom SPARQL that returns a subgraph of the knowledge

[29] The second highest class is schema:DataType, whose subclasses are typically mapped to XSD datatypes.

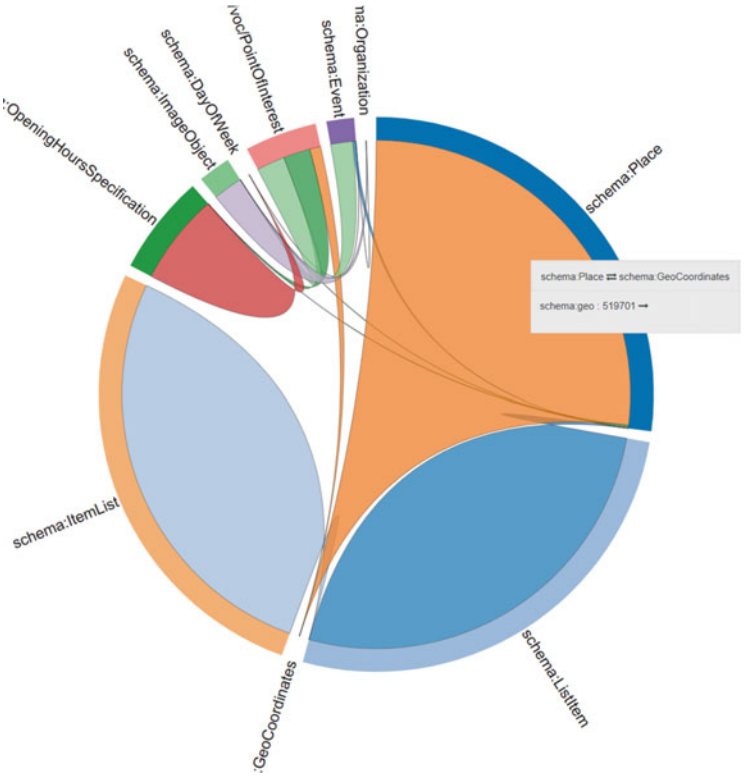


Fig. 19.9 Class hierarchy visualization of the German Tourism Knowledge Graph

graph. The visualization shows the incoming and outcoming edges of a node identified by an IRI or selected via SPARQL queries. Figure 19.10 shows the graph-based visualization of the instance in Fig. 19.1. Initially, only 1-hop connections are shown. However, the visualization can be configured, and the users can interact with the graph to expand further nodes.

19.4.4 Reasoning

GraphDB provides a reasoner based on W3C recommendations to infer new facts from the existing ones in a knowledge graph. The reasoner applies the entailment rules for different formalisms in a forward chaining manner. GraphDB has a

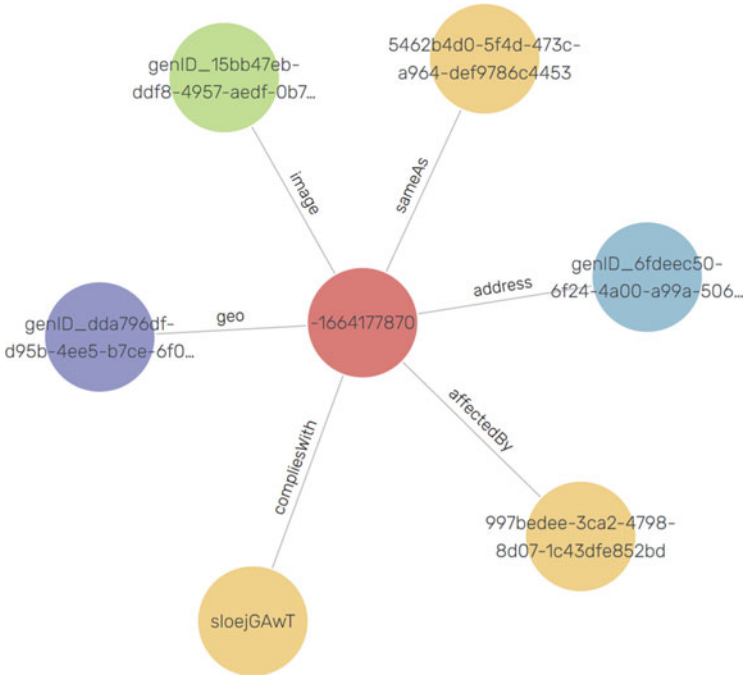


Fig. 19.10 An interactive graph visualization of the instance is represented in Fig. 19.1

materialization strategy, which means that inference rules are applied repeatedly to the explicit statements until no further inferred (implicit) statements are produced. Inferred statements are produced at load time, which indicates longer loading times but faster query times.[30]

GraphDB supports various reasoning profiles, which include RDF(S), RDFS-Plus,[31] and OWL2 profiles (Horst, DL, QL, RL). These profiles make different trade-offs and are suitable for different use cases. For the definition of DL, QL, and RL profiles, we refer the readers to Sect. 13.3. OWL-Horst (ter Horst 2005) is not part of the W3C recommendation for OWL2, but it is a profile that is used in triplestore implementations. It provides further restrictions on OWL RL to improve reasoning performance for large datasets.[32] For the German Tourism Knowledge Graph, the

[30] See also https://graphdb.ontotext.com/documentation/enterprise/introduction-to-semantic-web.html#introduction-to-semantic-web-reasoning-strategies

[31] RDF(S) semantics extended with some OWL constructs such as inverse, symmetric, and transitive properties.

[32] Both OWL-RL and OWL-Horst bring rules to OWL, but OWL-Horst only supports RDFS and some OWL constructs like *sameAs*, *equivalentClass*, *equivalentProperty*, *SymmetricProperty*, *TransitiveProperty*, *inverseOf*, *FunctionalProperty*, and *InverseFunctionalPropertry*. See the post from the GraphDB CEO on Stackoverflow for a detailed comparison. https://stackoverflow.com/questions/63163024/horst-pd-compared-to-owl2-rl

Table 19.8 Overview of the different hosting paradigms

Relational database	Document database	Graph database (triplestore)
Fixed schema.	No fixed schema.	No constraining, rigid schema (still they can have ontologies).
Schema adaptations require a change of the database schema.	Documents in a collection can have different properties; adding properties possible without affecting already existing documents.	Allows to easily integrate data using different schemas (ontologies); adding properties does not affect existing graphs as in RDB.
Data are organized in different tables.	Documents are organized in collections.	Triples can be organized in named graphs.
JOINs required to combine data from different tables.	Allows embedding documents; JOINs required when using identifiers to link to other documents.	JOINs are required, but they are handled differently than relational databases.
No native reasoning.	No native reasoning.	Native logical reasoning.

RDFS-Plus reasoning profile is used. RDFS-Plus is also a profile that is driven by real-world applications but not a W3C recommendation. It is more restricted than OWL-Horst, but it is enough to capture the subsumption reasoning for types and properties and reasoning for inverse and transitive properties (Allemang et al. 2020).

GraphDB contains two built-in "virtual" named graphs for the materialization of statements, namely, explicit and implicit named graphs. Explicit named graph contain only the triples that are explicitly loaded to an RDF dataset. Contrarily, the implicit named graph contain only the ones that are inferred. During the query time, it can be specified whether only explicit statements should be considered or explicit and implicit statements together. For instance, in the first scenario, a query to our graph in Fig. 19.1 for "All Local businesses" would not return any result; however, in the second scenario, we would retrieve all instances of *schema:Hotel* due to subsumption reasoning and the subclass relationship between *schema:Hotel* and *schema:LocalBusiness* types.

19.5 Summary

Hosting knowledge graphs comes with a variety of challenges due to the data model, size, and heterogeneity; the velocity of change; and the different points of view and access modalities for different use cases. In this section, we introduced different hosting paradigms and analyzed them in terms of their capability to host a knowledge graph. Table 19.8 shows an overview of three different paradigms and their advantages and disadvantages.

Relational databases fit well for relational data with tabular representation with a stable and consistent structure. They are ideal for transactional environments where data integrity has the utmost importance. There are different modeling strategies with different trade-offs for modeling knowledge graphs, such as statement tables or

class-centric and property-centric modeling. In any case, relational databases, due to their rigid structure, cannot properly address challenges such as the data model, heterogeneity, and the velocity of knowledge graphs. As an alternative, virtual knowledge graphs can be built via a layer on top of an existing relational database. However, this way works based on query rewriting; therefore, typically, it has limited support for reasoning and querying.

Another alternative is the *document model* paradigm, where instances in a knowledge graph are stored as documents in so-called collections. Instances can be hosted as nested objects or can be hosted in different collections and referenced from there. Since document stores do not have a rigid schema, they address many challenges of hosting knowledge graphs well. They can deal with heterogeneity and velocity; however, the graph nature of the data model is tricky to handle. The nested object approach makes it cumbersome to update the data. The referencing across collections converts the document store into a relational database in the worst case. Since document stores typically provide instant access to a document, they are still quite useful for use cases where instances in a knowledge graph must be injected into a Web page as semantic annotations.

Finally, *graph databases* provide an ideal way to host knowledge graphs as they allow flexible schemas and natively support the graph data model of knowledge graphs. There are two widespread types of graph databases for knowledge graphs, namely, property graphs and native RDF triplestores. Both provide mechanisms to address the presented challenges of hosting knowledge graphs quite well; they both provide a flexible data model, handle large sizes well, and provide mechanisms to contextualize knowledge graphs to support different points of view, such as native support for describing relationships in property graphs and named graphs and RDF-Star (see Sect. 13.1.1) for RDF triplestores.

RDF triplestores are undoubtedly the most widespread way of hosting knowledge graphs as there are many well-established implementations, and they are built on top of W3C standards. In addition, they provide native reasoning support, which is an important selling point for knowledge graphs. We illustrated their usage with the German Tourism Knowledge Graph and GraphDB.

References

Ali W, Saleem M, Yao B, Hogan A, Ngomo ACN (2021) A survey of RDF stores & SPARQL engines for querying knowledge graphs. VLDB J 31(3):1–26

Allemang D, Hendler J, Gandon F (2020) Using RDFS Plus in the wild. In: Semantic we for the working ontologist: effective modeling for linked data, RDFS, and OWL. Morgan & Claypool

Angles R, Gutierrez C (2008) Survey of graph database models. ACM Comput Surv 40(1):1–39

Ben Mahria B, Chaker I, Zahi A (2021) An empirical study on the evaluation of the RDF storage systems. J Big Data 8:1–20

Codd EF (1970) A relational model of data for large shared data banks. Commun ACM 13(6): 377–387

Connolly TM, Begg CE (2005) Database systems: a practical approach to design, implementation, and management. Pearson Education

Harrington JL (2016) Relational database design and implementation. Morgan Kaufmann

Karvinen P, Diaz-Rodriguez N, Grönroos S, Lilius J (2019) RDF stores for enhanced living environments: an overview. In: Enhanced living environments: algorithms, architectures, platforms, and systems. Springer, pp 19–52

ter Horst HJ (2005) Combining RDF and part of OWL with rules: Semantics, decidability, complexity. In: The Semantic Web–ISWC 2005: 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6–10, 2005. Proceedings 4, Springer, pp 668–684

Xiao G, Ding L, Cogrel B, Calvanese D (2019) Virtual knowledge graphs: an overview of systems and use cases. Data Intell 1(3):201–223

Xiao G, Lanti D, Kontchakov R, Komla-Ebri S, Guzel-Kalaycı E, Ding L, Corman J, Cogrel B, Calvanese D, Botoeva E (2020) The virtual knowledge graph system Ontop. In: The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II, Springer, pp 259–277

Chapter 20
Knowledge Assessment



We covered the knowledge generation and hosting processes and have, as a result, an established knowledge graph (Fig. 20.1). Unfortunately, we now start with the *real* work, the knowledge curation process. We work on improving the *quality* of our knowledge graph. In general, this includes three subtasks:

- *Knowledge assessment* that investigates the quality of our knowledge graph. Its intention is not to change or update the graph but instead to evaluate and guide the further knowledge curation process.
- *Knowledge cleaning* when a relevant error rate has been identified.
- *Knowledge enrichment* when relevant gaps are found in the graph.

If this is not the case, we can leave this circle and start to deploy the knowledge graph.

Now, let us dive into this important step of knowledge curation, i.e., let us discuss knowledge assessment.[1] Knowledge assessment describes and defines the process of *assessing the quality* of a knowledge graph. What is quality assessment? It tries to measure the *fitness for use*, i.e., whether knowledge graph complies with the user's need (Wang and Strong 1996).

Quality is assessed based on a set of *dimensions*. Each dimension has a set of *metrics***.** Each metric has a calculation function. For measuring quality quantitatively,[2] we need to define weights for dimensions and metrics and aggregate this to an overall judgment of the knowledge graph (see also Fãrber et al. 2018). For knowledge assessment, many researchers were inspired by the work on data quality. The data quality community typically organizes the quality dimensions in four categories (Wang and Strong 1996):

- *Intrinsic* – dimensions that can be measured only with the data at hand
- *Contextual* – dimensions that depend on the context of the user or task

[1] This chapter contains content derived from Angele et al. (2019) and Fensel et al. (2020).

[2] Whether it is possible to measure quality quantitatively, see Pirsig (1974, 1991).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_20

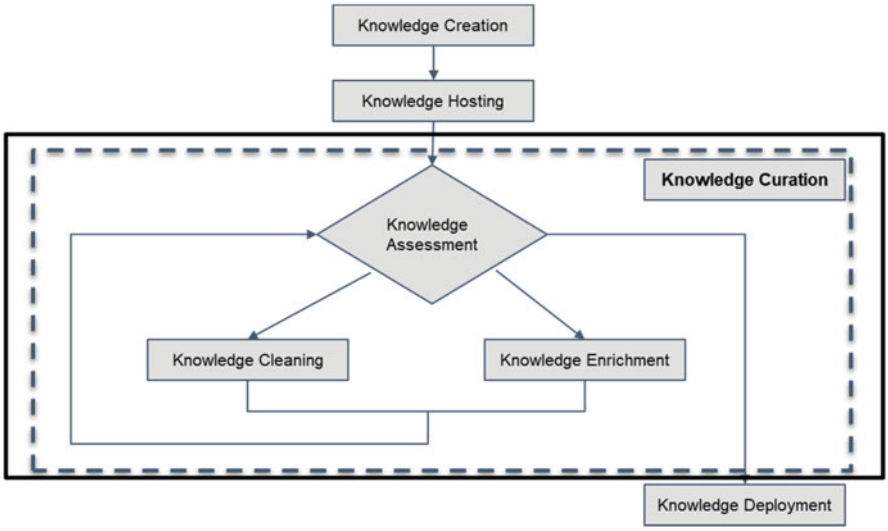


Fig. 20.1 The knowledge graph lifecycle

- *Representational* – dimensions that are related to the representation of the data
- *Accessibility* – dimensions regarding the level of accessibility of the data

Based on the literature (see also Strong et al. (1997), Wang (1998), Wang et al. (2001), Pipino et al. (2002), Batini and Scannapieco (2006), Zaveri et al. (2016), and Fãrber et al. (2018)), the quality dimensions we consider for knowledge graphs are:

1. Accessibility
2. Accuracy
3. Believability
4. Completeness
5. Concise representation
6. Consistent representation
7. Cost-effectiveness
8. Ease of manipulation
9. Ease of operation
10. Ease of understanding
11. Relevancy
12. Security
13. Timeliness
14. Traceability

Quality dimensions are defined with their respective criteria, metrics, and measurement functions.

- *Criteria* define certain characteristics of a knowledge graph with regard to a quality dimension.

- *Metrics* express the criteria in measurable terms.
- *Measurement functions* are metrics expressed in terms of functions that return certain values as a measurement for each metric.
- Task- and application-specific *weights for the dimensions and the metrics* they use.

In the following, we will discuss these 14 dimensions. Then we discuss a calculation approach for quality, sketch some approaches, and provide conclusions.

20.1 Quality Dimensions

We define 14 quality dimensions for knowledge graphs based on the literature cited above. Here, we only make brief informal definitions of the dimensions and their criteria and measurement functions, based on the definitions in Fãrber et al. (2018). For more dimensions, metrics, detailed discussion, and formal definitions, see Fãrber et al. (2018) and Zaveri et al. (2016).

Accessibility implies that a knowledge graph or parts of it must be available and retrievable and contain a license. Potential metrics and measurement functions are:

- *Availability of the knowledge graph*, measured by monitoring the ability of dereferencing URIs over a period of time.
- *Provision of an endpoint* evaluates whether an endpoint to access the knowledge graph is available. This metric can be calculated via a function that returns certain values depending on the type of the available endpoint (e.g., returns 1, if SPARQL endpoint is available, 0.5 if an HTTP API is available, and 0 if no endpoint is available).
- *Retrievable in RDF format* metric evaluates whether an RDF export dataset of the knowledge graph is available. This metric can be calculated via a function that returns a Boolean value (e.g., returns 1 if RDF export is available).
- *Support of content negotiation* evaluates whether content negotiation is supported by the knowledge graph. For instance, the knowledge graph returns the desired content type (e.g., Turtle, JSON-LD). This metric can be calculated via a function that returns a Boolean value (e.g., returns 1 if content negotiation is supported).
- *Containing a license* evaluates whether a knowledge graph contains a license under which the knowledge graph data may be used. This metric can be calculated via a function that returns a Boolean value (e.g., returns 1 if license information is available).

Accuracy defines the reliability and correctness (syntactically and semantically) of the data. This dimension is sometimes interchangeably used with correctness. Potential metrics and measurement functions are:

- *Syntactic correctness of property value assertions* evaluates whether the literal values are syntactically correct,[3] e.g., a property value is syntactically correct if it follows certain regular expressions (e.g., literals describing dates must follow a certain ISO standard). This metric can be measured via calculating the ratio of syntactically correct literal values to all literal values.
- *Semantic correctness of property value assertions* evaluates whether the meanings of triples are semantically correct, e.g., a value is semantically valid if the value is true. The measurement of this metric must be handled in two fronts, namely, the formal correctness and domain correctness. Formal correctness checks whether the property value assertions satisfy a formal specification (e.g., a set of SHACL constraints, an ontology). Domain correctness is about the semantic correctness of assertions in the domain they describe (e.g., if the given phone number of a hotel belongs to that hotel). The measurement of this metric can be done via checking a knowledge graph against formal specifications programmatically and against the "real world" via a canonical source or manually (e.g., by actually calling the hotel to see if the phone number is correct).

Believability, or trustworthiness, defines the degree to which the knowledge graph is regarded as "true, credible, real, trustable, unbiased, and verifiable." Potential metrics and measurement functions are:

- *Provenance information* evaluates the existence of provenance information in the knowledge graph, e.g., using Dublin Core or PROV-O for storing provenance information. The metric can be measured by a function that assigns different scores for the different levels of provenance information (e.g., 0 for no provenance, 0.5 for the overall knowledge graph, and 1 for additional declaration for the statements).
- *Trustworthiness* evaluates how trustworthy the knowledge graph is regarding its curation. This metric can be calculated via a function that assigns different scores to the curation method of a knowledge graph (e.g., a high trustworthiness score if the knowledge graph is curated manually by a group of experts and a lower score if the knowledge graph is curated with automated methods).

Completeness refers to the degree to which a knowledge graph contains the necessary knowledge for a task or domain. In the (linked) data quality literature, the metrics for this dimension are typically defined in terms of whether the knowledge graph instantiates all types and relationships in an ontology or it is complete in comparison to a knowledge graph that is seen as golden standard. We see such an approach to completeness problematic, as it ignores the contextual nature of the completeness dimension. To simplify the matter, we propose two metrics (Şimşek et al. 2022):

[3] This metric can be easily extended to syntactic correctness of IRIs. In this case, at least for the class and property IRIs, the ontology used in the knowledge graph can be seen as a reference point to check whether IRIs have syntactical errors.

- *Domain coverage* refers to whether a knowledge graph is complete in terms of the requirements of a given domain. The metric can be calculated via a function that returns the ratio of an average number of properties used on the instances of a type to a number of all properties of that type defined by a domain expert. If there are multiple types involved, then an average score across all types can be calculated. This metric can be implemented with the help of the domain-specific patterns introduced in the knowledge creation process (Chap. 18).
- *Task coverage* refers to whether a knowledge graph is complete in terms of the requirements of an application. This metric can be calculated via a function that returns the ratio of successful queries to a knowledge graph to all queries by an application. The metric can be implemented with the help of examining the query logs of the knowledge graph.

Concise representation refers to how compactly the knowledge graph (schema and instance level) is represented.[4] Potential metrics and measurement functions are:

- *Avoiding blank nodes* evaluates the existence of triples that contain blank nodes. The calculation of these metrics can be either done in a discrete manner (i.e., whether a blank node exists) or can be done via calculating a ratio (e.g., the number of blank nodes used to a number of triples). Note that although using blank nodes may hinder conciseness, they may be also necessary for some use cases.
- *Avoiding RDF reification* metric evaluates the use of RDF reification. This metric can be calculated via checking whether an RDF reification method is used. Note that although reification may harm conciseness, it may be still necessary to use it (e.g., for provenance). Therefore, the evaluation of this metric can be more sophisticated, for example, with the consideration of the used reification method. Different reification methods have different space complexity (Hernandez et al. 2015).

Cost-effectiveness measures the total cost of using (or collecting) the data in a knowledge graph (adapted from the dimension in Wang and Strong (1996)). The only metric in this dimension is *cost of data.* The metric can be calculated by mapping the cost of data and the perceived usefulness of the data to a value between 0 and 1.

Ease of manipulation refers to how easy it is to manipulate and apply the knowledge graph to different tasks (Pipino et al. 2002). This dimension is mostly about the *documentation* metric, which can be calculated by checking the existence of documentations for manipulating the knowledge graph.

Ease of operation refers to how easily knowledge graph data can be updated, downloaded, and integrated with other knowledge graphs. This dimension can be seen as a more granular version of *ease of manipulation*.

[4] This dimension is also considered while measuring the interoperability of knowledge graphs. For example, skolemizing all blank nodes would make it easier to align with other knowledge graphs.

- *Updating metric* evaluates the possibility of updating the knowledge graph. It can be calculated via a function that returns a Boolean value.
- *Downloading metric* evaluates the possibility of downloading the knowledge graph in open standards, e.g., RDF. It can be calculated via a function that returns, e.g., 0 if it is not possible to download data from the knowledge graph, 0.5 if machine-processable data can be downloaded, and 1 if RDF data can be downloaded.
- *Integration metric* evaluates whether a knowledge graph can be easily integrated to other sources. One way to measure this could be checking if the knowledge graph uses standard vocabularies or follows other best practices for data publication.

Ease of understanding refers to how easy it is for humans to understand the knowledge graph (Fãrber et al. 2018). The potential metrics and their measurement functions are:

- *Self-descriptive URI* refers to whether self-describing URIs are used to identify resources, e.g., *https://dbpedia.org/resource/Innsbruck* vs *https://www.wikidata.org/wiki/Q1735*.
- *Various languages* evaluates the degree to which data are described in more than one language, for instance, property values of *rdfs:label* or *schema:name* in different languages.

Relevancy defines the level of applicability of the knowledge graph given a specific task or domain (Wang and Strong 1996). This dimension is highly related to completeness, but not necessarily the same. A knowledge graph that is not complete for a domain and task may still be relevant. One potential metric for this dimension is "support for ranking statements." This metric evaluates whether the knowledge supports a mechanism to rank property value assertions for an instance. This would allow us to distinguish between the most relevant property value assertions and the other ones. The metric can be calculated via a function returning a Boolean value (e.g., returns 1, if the knowledge graph supports a ranking system).

Security indicates how access to the knowledge graph is restricted (Wang and Strong 1996) to maintain its integrity and prevent its misuse (Zaveri et al. 2016). A potential metric for the security dimension is the *usage of digital signature* which evaluates whether statements are digitally signed, e.g., to verify the integrity of the data and identity of the publisher of the data.

Timeliness refers to the up-to-datedness of the data used for a knowledge graph. The potential metrics and their measurement functions are:

- *Frequency of updates* evaluates how often the knowledge graph is updated. This metric can be measured in different ways, for example, via a function that maps certain range of frequencies to numerical values between 0 and 1.
- *Support for validity period of statements* evaluates whether the knowledge graph supports the specification of validity periods for the statements. This metric can be calculated via a function that returns a Boolean value (e.g., returns 1 if there is a mechanism for specifying the validity periods and 0 otherwise).

Traceability refers to the degree to which "data is well documented, verifiable, and easily attributed to a source" (Wang and Strong 1996). This dimension is highly related to provenance metadata. A potential metric for this dimension is the *representation of provenance information*. The metric evaluates the degree to which the provenance information is supported for a knowledge graph. The metric can be calculated via a function that maps different levels of provenance support to a numerical value between 0 and 1.

20.2 Calculating Quality Score

The calculation of the overall quality score for a knowledge graph can be summarized in three steps:

1. Deciding on dimension weights: Not every dimension may be relevant or as important as others for a given domain or task. Therefore, we decide weights on each dimension we want to use in our assessment.
2. Deciding on metric weights: Similarly, not every metric may have the same importance within a dimension. Therefore, we decide on the weights of each metric for the relevant dimensions.
3. Calculating an aggregated quality score: After the weights are determined, an aggregated quality score for the knowledge graph is calculated as described in the following.[5]

Given a knowledge graph k, the weighted aggregate score of the i^{th} dimension for $val(k, d_i(k))$ is calculated as

$$val(k, d_i(k)) = \sum_{j=1}^{p_i} \alpha_{i,j} * m_{i,j}(k)$$

where

- p_i is the number of metrics applied to the i^{th} dimension
- $\alpha_{i,j}$ is the weight of the j^{th} metric of the i^{th} dimension with $\sum_{j=1}^{p_i} \alpha_{i,j} = 1$
- $m_{i,j}(k)$ is the score of the j^{th} metric of the i^{th} dimension $m_{i,j}(k) \in [0, 1]$ for all $m_{i,j}(k)$

When the scores for all dimensions are calculated, the weighted aggregated quality score for a knowledge graph $val(k)$ is *calculated as*

[5] Adapted from Şimşek et al. (2022)

$$val(k) = \sum_{i=1}^{n} \beta_i * val(k, d_i(k))$$

where

- n is the total number of dimensions considered
- β_i is the weight of the i^{th} dimension with $\sum_{i=1}^{n} \beta_i = 1$

20.3 Approaches and Illustration

Knowledge assessment is a task whose solutions are quite hard to generically implement. Definition of dimensions, metrics, and their measurement functions may be defined differently for different communities, and the implementation of measurement functions may depend on specific knowledge graphs. Moreover, the level of automation for the calculation may vary, as some metrics can be only measured via human intervention.

Considering these characteristics of the assessment task, the approaches can be viewed from three perspectives: crowdsourcing, generic frameworks that need to be configured and adapted for specific knowledge graphs, and specific frameworks that focus on a narrower set of dimensions (e.g., only accuracy).

Crowdsourcing-driven assessment approaches are particularly useful, when the input of experts is needed (e.g., the assessment of semantic validity of statements). An example crowdsourcing framework is presented by Acosta et al. (2013). The framework approaches the assessment task in three steps:

1. Create a schema for quality issues.
2. Ask domain experts to find quality issues.
3. Verify quality issues via paid microtasks.

Crowdsourcing approaches can be replaced or complemented via semi-automated approaches supported with tools. Some approaches are:

- LINK-QA (Gueret et al. 2012) that assesses quality in a knowledge graph using network metrics.
- Luzzu (Debattista et al. 2016) is a configurable and extendable quality assessment framework which we will discuss in more detail.
- RDFUnit (Kontokostas et al. 2014) generates test cases for validating graphs mostly focusing on accuracy and completeness dimensions.
- SDType and SDValidate (Paulheim and Bizer 2014) use statistical distribution for measuring the correctness of statements.
- Sieve (Mendes et al. 2012) provides means to assess datasets. It defines data quality indicators, scoring functions, assessment metrics, and aggregate metrics.
- Many SHACL and ShEx verifiers (see Gayo et al. (2017) for a list) target mainly accuracy and completeness dimensions by verifying RDF graphs against shapes.

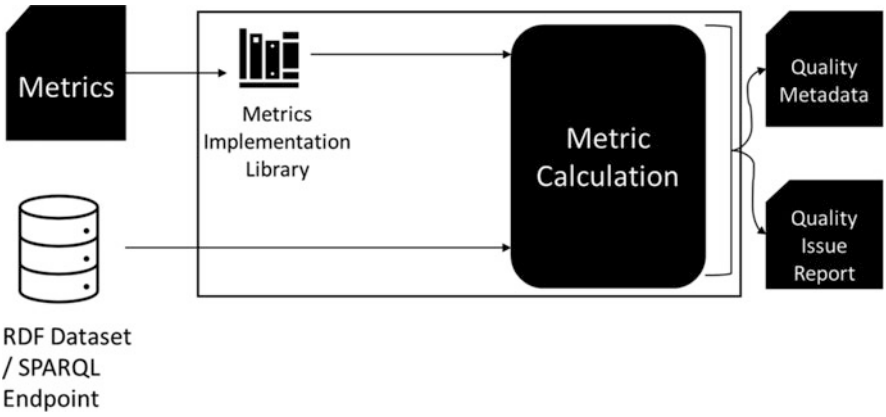


Fig. 20.2 The general architecture of Luzzu

As mentioned in the beginning of this section, due to the characteristics of the assessment task, the implementations must be configurable and flexible. Luzzu linked data quality assessment framework is an example of such a framework. The high-level process model is shown in Fig. 20.2.[6,7] The metric implementations in the metrics implementation library can be either done programmatically as a Java library or declaratively via a custom language offered by Luzzu as condition-action rules. The metric calculation module takes metric definitions and implementations as input as well as triples from an RDF dataset or retrieved via a SPARQL endpoint. For scalability, Luzzu supports streaming and parallel processing during metric calculations. Once a metric calculation is done, Luzzu produces two outputs, quality metadata that summarizes the assessment process and contains quality scores for different dimensions and metrics. The quality issue report contains more granular information about the quality issues found in the assessed RDF dataset. Both outputs are semantically described, therefore can be reused by other applications, and can be queried with SPARQL. This allows storing assessment knowledge together with the dataset which can be then used, e.g., to rank different datasets based on their assessment results.

Now, let us illustrate how a metric can be calculated concretely on a knowledge graph (Şimşek et al. 2022). Assume that we are assessing the syntactic correctness of "property value assertions" metric in the accuracy (correctness) dimension in the knowledge graph in Fig. 20.3.

The measurement function for this metric is the ratio of the number of correct literals to the number of all literals. In Fig. 20.3, there are a total of five literals (i.e., the total number of rectangles in the figure). During the assessment, we identify that only one of the geocoordinates, the value of *schema:latitude* property (4926178),

[6] https://github.com/Luzzu/Framework

[7] https://www.slideshare.net/jerdeb/data-quality-123463530

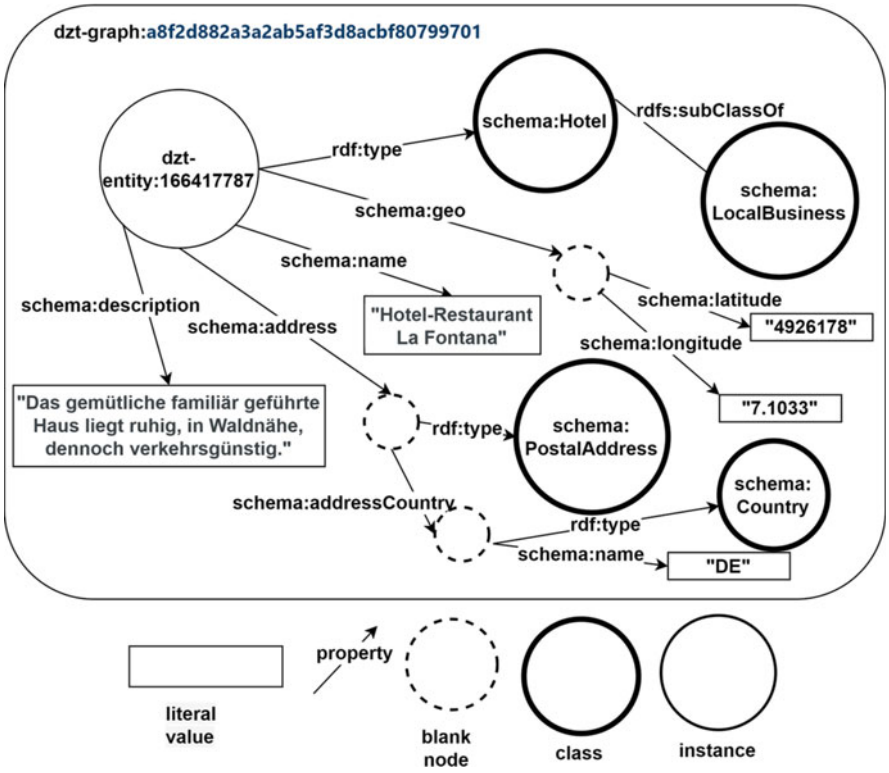


Fig. 20.3 An excerpt from the German Tourism Knowledge Graph

does not fit the syntactic structure of the WGS84[8] system in decimal degrees. The decimal coordinate system accepts values in [−90, 90] for latitude and in [−180, 180] for longitude. The latitude value 4926178 is outside of the given range. With four remaining correct literals, the quality score for our metric is $4/5 = 0.8$.

20.4 Summary

Knowledge graphs with poor quality cost organizations significantly. To improve the quality of knowledge graphs, their quality status must be assessed. The data quality community produced a large set of quality dimensions and metrics. These dimensions and metrics are vastly adopted by the Semantic Web and knowledge graph community.

[8] https://en.wikipedia.org/wiki/World_Geodetic_System

Among many dimensions in the literature, we presented 14 dimensions and their potential metrics for assessing knowledge graphs. Clearly, there are overlaps and trade-offs between different dimensions. For example, usage of RDF reification may harm the concise representation dimension, but it is also necessary for attaching provenance data for the traceability dimension. Therefore, the first step for assessment is deciding on the weights of the dimensions based on their importance, where irrelevant dimensions get a zero weight. With a similar motivation and in a similar fashion, the metrics are also weighted per dimension.

The assessment process gives insights about the overall quality of the knowledge graph in different dimensions. It does not aim to find individual errors. However, tools for finding specific errors can be used for some parts of the calculation of the overall quality score. We mentioned some approaches and tools for quality assessment and introduced a configurable and extendable framework called Luzzu. Later in Sect. 23.1.3, we will present a reference implementation for the quality score calculation process presented in this chapter.

Among 14 dimensions, we will focus particularly on 2 during the knowledge graph lifecycle: accuracy (correctness) and completeness. A low correctness score may trigger a *knowledge cleaning* process. A low completeness score may trigger a *knowledge enrichment* process.

References

Acosta M, Zaveri A, Simperl E, Kontokostas D, Auer S, Lehmann J (2013) Crowdsourcing linked data quality assessment. In: The Semantic Web–ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21–25, 2013, Proceedings, Part II 12, Springer, pp 260–276

Angele K, Fensel D, Harth A, Huaman E, Panasiuk O, Şimşek U (2019) Ein URI konzept für den deutschen tourismus. Technical report. https://www.sti-innsbruck.at/sites/default/files/uri-concept/uri-concept.pdf

Batini C, Scannapieco M (2006) Data quality: concepts, methodologies and techniques. Data-centric systems and applications. Springer

Debattista J, Auer S, Lange C (2016) Luzzu—a methodology and framework for linked data quality assessment. J Data Inf Qual 8(1):1–3

Fãrber M, Bartscherer F, Menne C, Rettinger A (2018) Linked data quality of DBpedia, Freebase, OpenCyc, Wikidata, and Yago. Semant Web 9(1):77–129

Fensel D, Simsek U, Angele K, Huaman E, Kärle E, Panasiuk O, Toma I, Umbrich J, Wahler A (2020) Knowledge graphs. Springer

Gayo JEL, Prud'Hommeaux E, Boneva I, Kontokostas D (2017) Validating RDF data. Synth Lect Semant Web Theory Technol 7(1):1–328

Gueret C, Groth P, Stadler C, Lehmann J et al (2012) Assessing linked data mappings using network measures. ESWC 7295:87–102

Hernandez D, Hogan A, Krötzsch M (2015) Reifying RDF: what works well with Wikidata? SSWS@ISWC 1457:32–47

Kontokostas D, Westphal P, Auer S, Hellmann S, Lehmann J, Cornelissen R, Zaveri A (2014) Test-driven evaluation of linked data quality. In: Proceedings of the 23rd international conference on World Wide Web, Seoul, Korea, April 7–11, pp 747–758

Mendes PN, Mühleisen H, Bizer C (2012) Sieve: linked data quality assessment and fusion. In: Proceedings of the 2012 joint EDBT/ICDT workshops, pp 116–123
Paulheim H, Bizer C (2014) Improving the quality of linked data using statistical distributions. Int J Semant Web Inf Syst 10(2):63–86
Pipino LL, Lee YW, Wang RY (2002) Data quality assessment. Commun ACM 45(4):211–218
Pirsig RM (1974) Zen and the art of motorcycle maintenance. William Morrow and Company
Pirsig RM (1991) Lila: an inquiry into morals. Bantam Books
Şimşek U, Kärle E, Angele K, Huaman E, Opdenplatz J, Sommer D, Umbrich J, Fensel D (2022) A knowledge graph perspective on knowledge engineering. SN Comput Sci 4(1):16
Strong DM, Lee YW, Wang RY (1997) Data quality in context. Commun ACM 40(5):103–110
Wang RY (1998) A product perspective on total data quality management. Commun ACM 41(2): 58–65
Wang RY, Strong DM (1996) Beyond accuracy: what data quality means to data consumers. J Manag Inf Syst 12(4):5–3
Wang RY, Ziad M, Lee YW, Wang Y (2001) Data quality
Zaveri A, Rula A, Maurino A, Pietrobon R, Lehmann J, Auer S (2016) Quality assessment for linked data: a survey. Semant Web 7(1):63–93

Chapter 21
Knowledge Cleaning



Knowledge cleaning consists of two significant subtasks: *error detection* and *error correction*. In a knowledge graph, various types of errors can be detected, such as syntactic and semantic errors. Semantic errors can be further classified into errors regarding formal semantics (detected via verification) and errors regarding the described domain (detected via validation). Error correction involves deleting, modifying, or adding[1] assertions at a knowledge graph.

In this chapter, we strictly distinguish between ABox and TBox. The TBox is defined by an external standardization process (like schema.org) that provides terminological knowledge that should be applied to model facts in a certain domain. We assume the TBox is the golden standard and locate errors in the ABox.

Throughout this chapter, we will use an excerpt from the German Tourism Knowledge Graph to explain different error types and potential corrections. We will introduce different errors to the excerpt shown in Fig. 21.1 and provide fixes for them. The same examples will also be used for a more holistic illustration of error detection and correction at the end of the chapter.

In the remainder of this chapter, we will first introduce different types of errors with examples that can occur in knowledge graphs. Afterward, we will present some error detection and error correction methods. Before we summarize, we give an illustrative example using the German Tourism Knowledge Graph.[2]

[1] Addition of assertions already brings us in the field of knowledge enrichment, which we will cover in the next chapter.

[2] See also Fensel et al. (2020).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_21

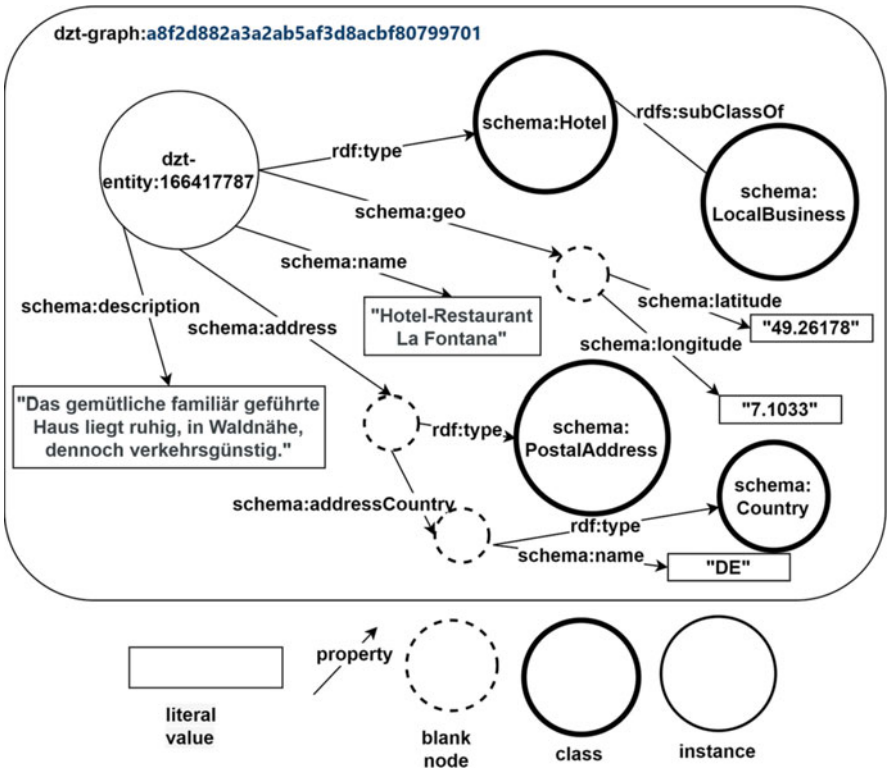


Fig. 21.1 An excerpt from the German Tourism Knowledge Graph as running example

21.1 Error Types

Before dealing with error detection, we will define what errors look like. We will consider three error sources:

- Wrong *instance assertions* (triples with *rdf:type* in the predicate position)
- Wrong *equality assertions* (triples with *owl:sameAs* in the predicate position)
- Wrong *property value assertions* (triples with an arbitrary property in the predicate position)

In the following, we will introduce possible errors and their fixes in these three conceptual categories. Hereby, these errors can be syntactical, semantic with regard to a formal specification, or due to semantically wrong statements in the domain of discourse.

Error:
dzt-entity:1664 17787 rdf:type schema:Hotel .

Correction:
dzt-entity:166417787 rdf:type schema:Hotel .

Fig. 21.2 A syntactic error in the instance assertion

Error:
dzt-entity:166417787 rdf:type schema:FastFoodRestaurant .

Correction:
dzt-entity:166417787 rdf:type schema:Place .

Fig. 21.3 A wrong instance assertion with a non-existing type

Error:
dzt-entity:166417787 rdf:type schema:Event .

Correction:
dzt-entity:166417787 rdf:type schema:Hotel .

Fig. 21.4 A wrong type assertion with regard to the domain of discourse

21.1.1 Wrong Instance Assertions

Instance assertions are of the form *a rdf:type b*, where *a* is an IRI or a blank node and *b* is a class. Possible errors are:

- Syntactic errors in the resource identifiers.
- Type does not exist in the vocabulary.
- Assertion is semantically wrong.

Syntactic errors in the resource identifiers occur when there are false or missing syntactic tokens. Figure 21.2 shows a syntactic error in the identifier. There is a space character (unencoded) in the IRI, which is not allowed.

In an instance assertion, a type that does not exist in an ontology may be used and lead to an error. Figure 21.3 shows an example of erroneous instance assertion, where the *schema:FastFoodRestaurant* type is instantiated, which does not exist in the schema.org ontology.

Finally, an instance assertion can be wrong regarding the domain of discourse. Figure 21.4 shows an example where the given IRI is specified as an instance of

schema:Event. Although there is syntactically and formally nothing wrong with this triple, in the domain, the specified instance is a hotel.

21.1.2 Wrong Equality Assertions

Equality assertions are of the form i_1 *owl:sameAs* i_2, where possible errors are:

- Syntactic errors in i_1 or i_2.
- The assertion is wrong in the domain of discourse.

Syntactic errors in instance identifiers (i_1 or i_2) occur when there are false or missing syntactic tokens. Figure 21.5 shows a syntactic error in one of the instance identifiers. There is a whitespace character (unencoded) in the IRI, which is not allowed.

Semantically wrong equality assertions occur when instance identifiers i_1 and i_2 are not "the same in the domain of discourse." For instance, the triples in Figure 21.6 show such an error where the specified instance identifiers do not refer to the same entity in the domain to which they belong. The *dzt-entity:166417787* refers to the Hotel La Fontana in Germany, whereas the Apart La Fontana is a hotel in Austria.

Error:
dzt-entity:1664 17787 owl:sameAs https://www.apart-lafontana.at/

Correction:
dzt-entity:166417787 owl:sameAs https://www.apart-lafontana.at/

Fig. 21.5 A same as assertion where the subject IRI is syntactically wrong

Error:
dzt-entity:166417787 owl:sameAs https://www.apart-lafontana.at/

Correction:
The assertion is deleted

Fig. 21.6 An equality assertion that is semantically wrong

21.1.3 Wrong Property Value Assertions

Property value assertions are of the form $p(i_1, i_2)$ where i_1 and i_2 are instance identifiers and p is a property. The following error types may occur:

- Syntactic errors in i_1, i_2, or p.
- p does not exist.
- Domain and range violations or the assertion is wrong in the domain.

Syntactic errors in instance identifiers occur in the same way as the previous two categories. Figure 21.7 exemplifies this with an arbitrary property value assertion.

Syntactic errors in property identifiers occur where there are false or missing syntactic tokens in a property IRI. Figure 21.8 shows that there is a typo in the *schema:address* property.

A related error type to the previous one is the case where the property is not defined in the vocabulary. To some extent, this is the same error as the previous one; however, they can still be distinguished via string similarity measures to understand whether there is a typo or the property does not exist in the vocabulary. Figure 21.9

Error:
dzt-entity:166 417787 schema:name „Hotel-Restaurant La Fontana“

Correction:
dzt-entity:166417787 schema:name „Hotel-Restaurant La Fontana“

Fig. 21.7 A syntactic error in a property value assertion

Error:
dzt-entity:166417787 schema:adress [
schema:addressCountry
[
schema:name „DE“
]
]
Correction:
dzt-entity:166417787 schema:address [
schema:addressCountry
[
schema:name „DE“
]
]

Fig. 21.8 A syntactic error in the property IRI *schema:address*

Error:
dzt-entity:166417787 schema:hasGeo [
schema:latitude 49.26178;
schema:longitude 7.1033
]

Correction:
dzt-entity:166417787 schema:geo [
schema:latitude 49.26178;
schema:longitude 7.1033
]

Fig. 21.9 The property *schema:hasGeo* does not exist in the schema.org vocabulary

Error:
ex:Törggelen schema:containedInPlace dzt-entity:166417787.
ex:Törggelen rdf:type schema:Event .

Correction:
ex:Törggelen schema:containedInPlace dzt-entity:166417787.
ex:Törggelen rdf:type schema:Place .

Fig. 21.10 A domain violation in a property value assertion

demonstrates this with a value assertion for the property *schema:hasGeo* which does not exist in the schema.org vocabulary.[3]

Domain and range violations are errors regarding a formal specification, in that case, regarding a given ontology or integrity constraints. Figure 21.10 shows a property value assertion that violates the domain of the *schema:containedInPlace* property. Assume that there is a *schema:Event* instance integrated into the knowledge graph and connected to the hotel represented with *dzt-entity:166417787* via the *schema:containedInPlace* property. This would cause a domain violation as the *schema:containedInPlace* is expected on type *schema:Place*.

Finally, a property value assertion can be semantically wrong regarding the domain which the statements describe. For example, the statements in Fig. 21.11 are from a formal point of view correct; however, they do not reflect the truth in the domain they describe. The hotel described in Fig. 21.1 is not located in Austria.

[3] This can be seen as a violation of a formal specification, rather than a syntactical error.

Error:
dzt-entity:166417787 schema:address [
schema:addressCountry
[
schema:name „AT“
]
]
Correction:
dzt-entity:166417787 schema:address [
schema:addressCountry
[
schema:name „DE“
]
]

Fig. 21.11 A semantically wrong property value assertion

21.2 Error Detection and Correction

Before we can correct errors, we must find them. Manual detection of errors can be extremely tedious. In this section, we present various semi-automated approaches to detect and potentially correct errors in a knowledge graph. We classify these approaches into three categories, namely:

- *Syntactical processing* (e.g., LOD Laundromat (Beek et al. 2014; Hemid et al. 2019))
- *Statistical methods* (Hellerstein 2008; Paulheim and Bizer 2013, 2014)
- *Logical and knowledge-based methods* (Ma et al. 2014; Papaleo et al. 2014; Chu et al. 2013, 2015; Dallachiesa et al. 2013; Rekatsinas et al. 2017; Rula et al. 2019)

In the following, we will briefly explain these approaches with examples from the literature.

21.2.1 Syntactical Processing

Syntactic errors in knowledge graphs are typically detected via RDF parsers, which we have already covered briefly in Chap. 13. There are also more advanced frameworks that detect (and potentially also correct) syntactic errors. We explain how such a process works with the *LOD Laundromat* tool[4] (Beek et al. 2014) as an

[4] https://github.com/LOD-Laundromat/LOD-Laundromat

example. The tool provides scalable, fast, and automated syntactic cleaning of Linked Open Data and knowledge graphs. The LOD Laundromat works in the following ten steps:

1. *Collect URLs that denote dataset dumps*: URLs for different dataset dumps are collected from pre-determined catalogs.
2. *Grouping of collected URIs for processing:* URLs from the same host are grouped and not processed in parallel to prevent issues regarding the request constraints of specific servers.
3. *Communicate with the hosting server:* Data dumps are retrieved with HTTP (S) requests with common RDF serialization content types.
4. *Unpack archived data:* Archived data dumps are unpacked.
5. *Guess serialization format:* Serialization format is efficiently guessed with the help of the syntactic characteristics of the file (e.g., XML-based syntax vs Turtle-based syntax).
6. *Identify and mitigate syntactic errors:* Common syntax error types are identified, and several heuristics are used to mitigate them. These error types include and are not limited to:

   (a) Bad encoding
   (b) Undefined IRI prefixes
   (c) Missing end-of-statement characters
   (d) Non-escaped illegal characters inside IRIs
   (e) Multiline literals in serialization formats that do not support them
   (f) Non-matching tags
   (g) IRIs that do not appear between angular brackets (Turtle-based syntax)
   (h) File ends with a partial triple (most likely an error while splitting an RDF document)

7. *Deduplicate statements:* This step eliminates syntactic redundancies in a data dump. For example, a numerical literal may be given for the same subject and property as "0.01" and "0.01000."
8. *Save RDF in a canonical serialization format:* The clean data is saved in N-Triples format.
9. *Use VoID metadata to find other datasets to clean:* The Vocabulary of Interlinked Datasets (VoID)[5] is an RDFS ontology for attaching metadata to an RDF dataset, typically to support Linked Data discovery. VoID allows to specify to which datasets a given RDF dataset contain links (e.g., via owl: sameAs). By using the links to different datasets, LOD Laundromat tries to recursively collect different data dumps to clean.
10. *Consolidate and disseminate data with error statistics:* The metadata about the cleaning process is published with the data to give indications to data publishers to prevent errors in the future.

[5] https://www.w3.org/TR/void

Syntactical errors are generally "low-hanging fruits" in terms of error detection, as syntax rules are well-defined. They are mostly the first step in detecting and correcting semantic errors, as a syntactically wrong statement may also cause semantic issues. The tools for syntactic error detection typically only focus on RDF syntax; however, there can be other syntactic errors, as we mentioned in Sect. 21.1.

21.2.2 Statistical Methods

In some cases, errors, particularly semantic errors, can be detected by checking the outliers (Hellerstein 2008). In statistics, outliers are extreme deviations among the observed data and may indicate errors. There are various outlier detection methods, like probabilistic and statistical modelling, linear regression models, and information theory models.

A fundamental descriptive statistical notion is the standard deviation. A datapoint is usually considered to be an outlier if its distance to the mean is at least two times the standard deviation. Standard deviation of a population is defined as follows[6]:

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

where μ is the mean of the population, N is the size of the population, and x_i is an individual element of the population.[7] As an example, take the following temperature measurements in Celsius as population: *[−22, −17, −25, −20, −5, −999, −25, −22, −20, −17, −5]*.

The standard deviation of this population is roughly 282.15, while the mean is −107. We check whether any value is either below $-107 - (2 \times 282.15) = -671.3$ or above $-107 + (2 \times 282.15) = 457.3$. *−999* does not fall into this interval; it is an outlier.[8] There are also certain cases where outliers do not necessarily indicate errors. Consider the histogram of cities in Austria by population in Fig. 21.12. While all datapoints are between 0 and 500000, there is one datapoint in the 1750000–2000000 range. Although this distribution indicates a significant outlier, this does not indicate any error as that data point represents *Vienna*.

As seen from the previous example, any knowledge cleaning method is to be used with caution. Human intervention is not off the table. Many proposed tools like

[6] There is a slight difference between population and sample standard deviation. In sample standard deviation, the division is by *N-1*, where *N* is the sample size.

[7] Note that σ^2 gives the *variance* of a dataset.

[8] Statistical methods can be also supported with heuristics. For example, our knowledge about temperature confirms that this is some sort of error as the lowest limit of the thermodynamic temperature scale is −273.15 Celsius.

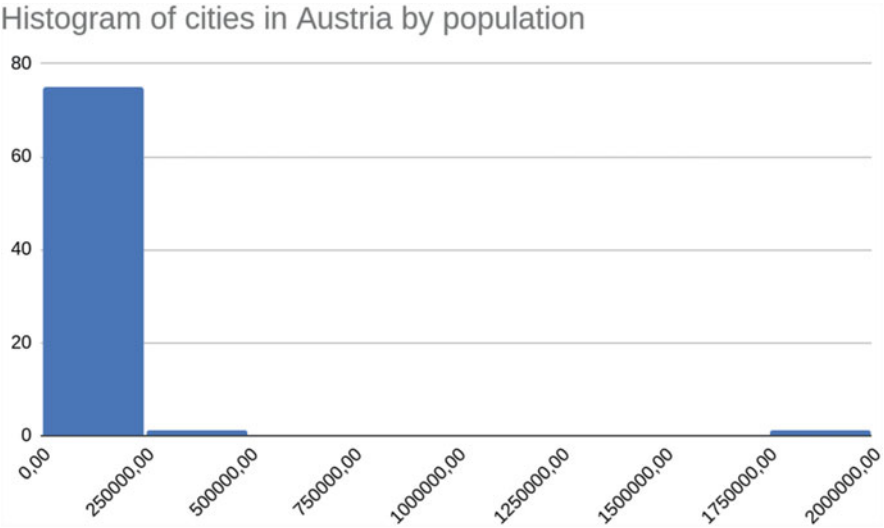


Fig. 21.12 A histogram of cities in Austria by population

HoloClean (Rekatsinas et al. 2017) and *KATARA* (Chu et al. 2015) are (weakly) supervised to mitigate any false assumptions because of statistical methods.

There are many other advanced methods implemented in software libraries. These are including but are not limited to *density-based spatial clustering of applications with noise*,[9] *isolation forests*,[10] *minimum covariance determinant*,[11] *local outlier factor*,[12] and *one-class support vector machines.*[13]

21.2.3 Logical and Knowledge-Based Approaches

Logical and knowledge-based reasoning can be used to find whether a set of formulas is violated by the knowledge graph. Since the languages used for modelling knowledge graphs typically have logical foundations, logic can be used to detect errors. Knowledge-based error detection makes use of explicit knowledge from different knowledge bases and humans. The border between logical reasoning-based methods and knowledge-based methods may be blurry. Obviously, the

[9] https://en.wikipedia.org/wiki/DBSCAN

[10] https://en.wikipedia.org/wiki/Isolation_forest

[11] Example implementation in scikit: https://scikit-learn.org/stable/modules/generated/sklearn.covariance.MinCovDet.html

[12] https://scikit-learn.org/stable/modules/generated/sklearn.covariance.MinCovDet.html

[13] Example implementation in scikit: https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html

knowledge-based methods also use logical reasoning. However, the focus is on inference rules and axions. Knowledge-based reasoning includes large amounts of explicit knowledge that is externally defined like a knowledge graph. In this section, we will introduce the following methods for error detection:

- Disjointness axioms (Ma et al. 2014)
- Detection of invalid sameAs statements (Papaleo et al. 2014)
- Functional dependencies (Chu et al. 2013)
- SHACL data shapes[14] (see Sect. 13.2.2 for details)

21.2.3.1 Disjointness Axioms

The disjointness axioms work based on the declaration that one class is disjoint with another class. In consequence, there *must not* be shared instances. By defining two types disjoint, for example, *schema:Place owl:disjointWith schema:Person*, we declare that an instance of *schema:Place* must not also be of type *schema:Person*. Using disjointness axioms is useful to detect wrong instance assertions in knowledge graphs by simply looking, for instance, assertions that violate such statements. However, very few knowledge graphs implement this (Ma et al. 2014). The reason for that is that many knowledge graphs are modelled with languages like OWL, which uses description logic as underlying logical formalism. Description logic does not have CWA or UNA. This would require an explicit statement of all disjointness axioms, and manual generation of disjointness axioms in large knowledge graphs may not be feasible. An alternative to this is to use a formalism that has CWA and UNA, such as F-logic.

21.2.3.2 Detection of Invalid sameAs Statements

Functional properties and local completeness can be used to validate *sameAs* statements (Papaleo et al. 2014). A functional property p can be expressed as $p(x, y) \wedge p(x, z) \rightarrow y \equiv z$. This means if p is a functional property, and an instance has two values for that property, these two values are the same. The following rule is applied to every functional property p where the object is a literal.

$$sameAs(x, y) \wedge p(x, l_1) \wedge p(y, l_2) \rightarrow synVals(l_1, l_2)$$

If x and y are the same, their values for p are semantically the same, where predicate *synVals* holds true, if two literals represent the same value.[15,16]

[14] https://www.w3.org/TR/shacl/

[15] Note that that may lead to an inconsistency under D-entailment, if l_1 and l_2 have incompatible lexical-to-value mappings (see also Sect. 14.2.2.2).

[16] There can be alternative implementations of *synVals*, e.g., based on string similarity and clustering. See Papaleo et al. (2014).

The following rule is applied to every functional property p where the object is not a literal.

$$sameAs(x, y) \Lambda\, p(x, z_1) \Lambda\, p(y, z_2) \rightarrow sameAs(z_1, z_2)$$

If x and y are the same, their objects for p should be semantically the same.

The following rule is applied to every *inverse* functional property p where the object is not a literal.

$$sameAs(x, y) \Lambda\, p(s_1, x) \Lambda\, p(s_2, y) \rightarrow sameAs(s_1, s_2)$$

If x and y are the same, their subjects for p should be semantically the same.

A local completeness (LC) rule specifies that the description of a resource is complete (closed) only for a subset of the domain, for a given ontology. For a locally complete subject x with a property value o for a property p and a *sameAs* specification between x and another resource y, the following rule applies.

$$sameAs(x, y) \Lambda\, p(x, o) \rightarrow p(y, o)$$

If the predicate p is local complete (closed), then we can detect inconsistencies since we can infer negative facts in case a fact involving p does not exists.

The disjointness, functional property, and local completeness rules are just examples of how logical reasoning can be used to detect errors. Naturally, the full expressivity of the logical formalism and domain-specific rules can be used to detect semantic errors with regard to formal specifications.

21.2.3.3 Functional Dependencies

Functional dependencies (Chu et al. 2013) indicate a certain relationship between properties, where one property uniquely determines another. Functional dependencies have been widely used in relational data cleaning and can also be applied to knowledge graphs. An example of two functionally dependent properties can be postal codes and cities. Assume an instance of *PostalAddress* with property values for *postalCode* and *city*. Such a functional dependency can be specified in FOL syntax as follows:

$$\forall a, a', c, c', p(PostalAddress(a) \Lambda\, city(a, c) \Lambda\, postalCode(a, p)$$

$$\Lambda\, PostalAddress(a') \Lambda\, city(a', c') \Lambda postalCode(a', p) \rightarrow c = c')$$

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix schema: <https://schema.org/> .
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:CityShape a sh:NodeShape;
        sh:targetClass schema:City;
        sh:property [
                sh:path schema:postalCode;
                sh:datatype xsd:string;
                sh:minCount 1;
                sh:pattern: "^\\d{4}$"
                ].
```

Fig. 21.13 A SHACL shape to constraint instances of *schema:City* type

It expresses the rule that for all postal addresses with the same postal code, the city values are equal. In other words, the postal code uniquely determines a city. Functional dependencies are useful but may become rapidly too many to be defined.

21.2.3.4 SHACL Data Shapes

Shape constraints define "a shape" that a certain part of a knowledge graph should fit. A shape can apply to, for example, an instance of a certain type or the values of a certain property. Figure 21.13 shows a SHACL[17] shape constraint that applies to all instances of the type *schema:City*. The shape specifies that each *schema:City* instance must have a *schema:postalCode* property value and the values of the schema:*postalCode* property must fit the given pattern.

21.2.4 A Few Frameworks for Error Detection and Correction

To demonstrate what else can be done using knowledge, we quickly mention selected tools: HoloClean (Rekatsinas et al. 2017), KATARA (Chu et al. 2015), and TISCO (Rula et al. 2019).

HoloClean is an error detection and correction tool. It uses an external dataset and rules to match the property values in a knowledge graph with the external dataset. The property values of the external dataset replace the ones in the evaluated dataset

[17] We covered the SHACL language, tools, and applications extensively in Chap. 13.

to correct the erroneous property value assertions. For example, if, in the evaluated dataset, Innsbruck's postal code is "9999" and the external dataset's postal code for Innsbruck is "6020," HoloClean will favor the latter. This implies that HoloClean assumes the knowledge in the external source is the source of truth. HoloClean combines statistical and logical approaches to identify errors:

- Integrity constraints.
- Quantitative statistics (outlier detection).
- Using matching rules allows for comparing the data with provided knowledge.

KATARA is a data-cleaning tool powered by external knowledge sources and crowdsourcing. It aims to identify wrong data and provides suggestions for repair by discovering patterns in tables based on a trusted external knowledge source. The discovered patterns are validated via a crowdsourcing process, and the validated patterns are used to identify errors and suggest repairs.

TISCO is a tool for identifying errors in a knowledge graph that occur due to temporal changes in the validity of facts. The most important task TISCO tackles is to determine the temporal scope of facts. It uses a three-phase algorithm for mapping facts to sets of time intervals[18]:

1. *Temporal evidence extraction*: Extracts information for a given fact from the Web and DBpedia. The main purpose is to obtain a set of vectors for each given fact. Each vector contains the subject, predicate, object, year, and number of occurrences of that fact in that year.
2. *Matching:* Computes an interval-to-fact significance matrix associated with a fact. This step tries to assign each fact to a time interval given the extracted vectors from the previous steps and a set of matching functions. The columns and rows of the matrix represent chronologically ordered years. Each cell in the matrix represents a significance score of the given fact for a time interval between the years represented by the row and the column.
3. *Selection and reasoning:* Remember that the whole point of TISCO is to assign facts to time intervals. Such an assignment is valuable, for example, when we want to find the correct values for a property of a subject for certain time intervals. For example, consider a triple**:** *Cristiano_Ronaldo :playedFor :Sporting_Lisbon.* We want to identify between which years this fact was correct. At this stage of the algorithm, we have a significance matrix for each fact for a given interval. Now, the algorithm takes the significance matrices of the facts that are about the same subject and determine the scope of those facts about that subject, i.e., when did Cristiano Ronaldo play for Sporting Lisbon and when he played for Real Madrid. Different methods are used in combination to determine the time scope for a fact:

   (a) *Neighbor-x function:* Selects a set of intervals whose significance score is close to the maximum significance score in the significance matrix. This function takes a significance matrix as input and a predefined score range

[18] See the original paper (Rula et al. 2019) for a more detailed example.

whose upper bound is the maximum significance score in the matrix and whose lower bound is a defined threshold *x*. It returns all intervals (matrix indices) where the significance score is in that predefined range.

(b) *Top-k function:* Selects the best (with the highest significance score) *k* intervals among the results of the *neighbor-x* function, where *k* is an integer greater than 0.

(c) *Allen's interval algebra:* Since, in the end, we might have multiple (overlapping) intervals for a fact, these intervals need to be merged to obtain a single interval. Allen's algebra defines relations between intervals that are:

(i) Distinct because no pair of definite intervals can be related with more than one of these relationships.

(ii) Exhaustive because any pair of intervals must have one of these relationships.

(iii) There are a total of 13 basic relations for any pair of intervals defined, namely, precedes, preceded by, meets, met by, overlaps, overlapped by, finishes, finished by, contains, during, starts, started by, and equals.[19] Based on these algebraic relations, TISCO merges any pair of intervals for a fact. For example, an interval *a (2007–2012)* and an interval *b (2013–2015)* are merged into a new interval *c (2007–2015)*, which starts at the starting point of *a* and ends at the finishing point of *b*.

Based on the mapped intervals, it can be identified which value for a property of a subject was correct at which time interval.

21.3 Illustration: Cleaning the German Tourism Knowledge Graph

In this section, we apply some of the methods we introduced throughout the chapter to create a larger illustration of knowledge cleaning. We consider an excerpt of the graph in Fig. 21.1 which is taken from the German Tourism Knowledge Graph. It is demonstrated in Fig. 21.14, with an error introduced.[20]

First, we will use shape constraints to detect the error and then explain certain heuristics that can be used to correct the error.

[19] See https://www.ics.uci.edu/~alspaugh/cls/shr/allen.html for a good summary of Allen's interval algebra.

[20] The example is reprinted from Şimşek et al. (2022).

```
dzt-entity:166417787 a schema:Hotel;
        schema:geo [
                schema:latitude 4926178;
                schema:longitude 7.1033
        ];
        schema:address [
                schema:addressCountry [
                        schema:name "DE"
                ]
        ]
```

Fig. 21.14 An excerpt from the knowledge graph with an error introduced in the geolocation (in Turtle syntax, prefixes omitted)

21.3.1 Error Detection

Shape constraints, particularly with SHACL, are a widely used way to define constraints on a knowledge graph to assure integrity. Figure 21.15 shows a potential one to apply constraints on the German Tourism Knowledge Graph. The geocoordinates are expected to be in WGS84 decimal degrees format,[21] which means the latitude is in the range of [−90, 90] and the longitude is in the range [−180, 180].

When we apply this shape, the verification will detect a wrong property value assertion. The property shape with the path *schema:geo/schema:latitude* is violated as the value is not in [−90, 90].

21.3.2 Error Correction

After the error is detected, we can carry on with the correction. Since the error we caught is a semantic one, there may not always be a one-size-fits-all solution for correcting it. However, there are some heuristics we can use based on the properties causing the error. In this case, we already have the knowledge that the latitude values should be between *-90* and *90*. Additionally, we have the knowledge that the hotel is in Germany and the geographical boundaries of Germany are well-known.[22] We also know that the geocoordinates have the decimal datatype.

Given all this prior information, we can run an algorithm to put a decimal point in the right place to correct the latitude coordinate. The only option is putting the point after the first or the second digit, as anything after that would move the coordinate outside the expected range. When we try to put the point after the first digit, we

[21] https://en.wikipedia.org/wiki/World_Geodetic_System

[22] https://en.wikipedia.org/wiki/Geography_of_Germany

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix schema: <https://schema.org/> .
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:HotelShape a sh:NodeShape;
        sh:targetClass schema:Hotel;
        sh:property [
                sh:path (schema:geo schema:latitude);
                sh:datatype xsd:decimal;
                sh:minInclusive -90;
                sh:maxInclusive 90
                ];
        sh:property [
                sh:path (schema:geo schema:longitude);
                sh:datatype xsd:decimal;
                sh:minInclusive -180;
                sh:maxInclusive 180;
        ] .
```

Fig. 21.15 A SHACL shape to constrain the geocoordinates of hotels

obtain a formally correct coordinate; however, now, we cause another semantic error with regard to the domain as the coordinates now correspond to Nigeria *(4.926178, 7.1033)*. This leaves us with our last option, which would lead to the correct coordinates for the hotel, namely, *49.26178, 7.1033*.

A similar strategy can be applied to fix any geocoordinate that is wrongly specified in the knowledge graph.

21.4 Summary

Knowledge cleaning is the process of detecting and correcting erroneous statements in a knowledge graph. Naturally, errors can occur both at the schema level (TBox) and at the assertional level (ABox). Given the generally overwhelming size of the assertions compared to the size and complexity of a schema in a knowledge graphs, we focus on the assertions for the cleaning task.

In this section, we presented different error types and sources that can appear in a knowledge graph and the approaches to detect and correct them. Although we presented those strictly categorized, it is also clear that the borders between different types of errors can be blurry. The error types regarding instance and equality assertions may be seen as a specific kind of error in property value assertions. Moreover, it may not always be clear that an error is syntactic (e.g., a typo) or semantic (e.g., a property value is wrongly used). We may also encounter that

different error types are the causes of each other; for example, a wrong instance assertion may cause the violation of the domain or range of a property. Note that error detection assumes most of the time a closed-world assumption since otherwise, it would not be very straightforward to catch semantic errors due to formal specifications. For example, domain and range violations would simply lead to the inference of new types for the property values.

All these peculiarities make knowledge cleaning a hard task and an interesting research topic. The heterogeneity and interplay between error types require customized solutions, hopefully, automatized to some extent with the help of heuristics and statistical analysis.

We have seen that there are many approaches that target error detection and only some of them also include components for correction. Also, many of the cleaning tools are more generic than just targeting knowledge graphs.

It is safe to say that the knowledge cleaning problem is not solved yet and correcting especially domain-specific errors is particularly challenging. How can we verify that the phone number of a restaurant is correct without calling the restaurant? As touching the physical world is typically not very feasible, many solutions resort to their simplified representations in the cyber world, such as an existing dataset or the Web.

Knowledge cleaning is about correcting wrong facts but does not deal with the missing facts, unless we are adding new statements to correct the existing ones. In the next section, we will talk about knowledge enrichment, where the primary focus is to enrich knowledge graphs with new statements.

References

Beek W, Rietveld L, Bazoobandi HR, Wielemaker J, Schlobach S (2014) LOD Laundromat: a uniform way of publishing other people's dirty data. In: The Semantic Web–ISWC 2014: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19–23, 2014. Proceedings, Part I 13, Springer, pp 213–228

Chu X, Ilyas IF, Papotti P (2013) Holistic data cleaning: putting violations into context. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), IEEE, pp 458–469

Chu X, Morcos J, Ilyas IF, Ouzzani M, Papotti P, Tang N, Ye Y (2015) KATARA: reliable data cleaning with knowledge bases and crowdsourcing. Proc VLDB Endowment 8(12):1952–1955

Dallachiesa M, Ebaid A, Eldawy A, Elmagarmid A, Ilyas IF, Ouzzani M, Tang N (2013) NADEEF: a commodity data cleaning system. In: SIGMOD. ACM, New York, pp 541–552

Fensel D, Simsek U, Angele K, Huaman E, Kärle E, Panasiuk O, Toma I, Umbrich J, Wahler A (2020) Knowledge graphs. Springer

Hellerstein JM (2008) Quantitative data cleaning for large databases, vol 25. United Nations Economic Commission for Europe (UNECE), pp 1–42

Hemid A, Halilaj L, Khiat A, Lohmann S (2019) RDF doctor: a holistic approach for syntax error detection and correction of RDF data. In: Proceedings of KEOD 2019, pp 508–516

Ma Y, Gao H, Wu T, Qi G (2014) Learning disjointness axioms with association rule mining and its application to inconsistency detection of linked data. In: The Semantic Web and Web Science: 8th Chinese Conference, CSWS 2014, Wuhan, China, August 8–12, 2014, Revised Selected Papers 8, Springer, pp 29–41

Papaleo L, Pernelle N, Sais F, Dumont C (2014) Logical detection of invalid sameAs statements in RDF data. In: Knowledge Engineering and Knowledge Management: 19th International Conference, EKAW 2014, Linköping, Sweden, November 24–28, 2014. Proceedings 19, Springer, pp 373–384

Paulheim H, Bizer C (2013) Type inference on noisy RDF data. In: The Semantic Web–ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21–25, 2013, Proceedings, Part I 12, Springer, pp 510–525

Paulheim H, Bizer C (2014) Improving the quality of linked data using statistical distributions. Int J Semant Web Inf Syst 10(2):63–86

Rekatsinas T, Chu X, Ilyas IF, Ré C (2017) HoloClean: holistic data repairs with probabilistic inference. Proc VLDB Endowment 10(11):1190–1201

Rula A, Palmonari M, Rubinacci S, Ngonga Ngomo AC, Lehmann J, Maurino A, Esteves D (2019) TISCO: temporal scoping of facts. In: Companion Proceedings of the 2019 World Wide Web Conference, pp 959–960

Şimşek U, Kärle E, Angele K, Huaman E, Opdenplatz J, Sommer D, Umbrich J, Fensel D (2022) A knowledge graph perspective on knowledge engineering. SN Comput Sci 4(1):16

Chapter 22
Knowledge Enrichment



Knowledge enrichment is about improving the *completeness* of a knowledge graph. In the literature, the completeness of a knowledge graph is defined in different ways. For example, Fürber and Hepp (2011) measure it on schema and instance level given a specific use case, while Zaveri et al. (2016) suggest measuring it by comparing the total number of classes, properties, and assertions to a knowledge base that is seen as a gold standard.

No matter what the definition is, we can conclude two things about completeness: first, that completeness is about reducing the number of missing statements in a knowledge graph and second, that the definition of what is "missing" is contextual, meaning it is defined with regard to an external reference point (e.g., another knowledge source or a use case). The knowledge enrichment process typically consists of the following steps (see also Bleiholder and Naumann (2009)):

- Identification of additional data and knowledge sources
- Lifting the syntax of the chosen sources to the syntax used by the knowledge graph
- Alignment of the TBox of the knowledge graph with the schema of the external sources
- Integration of an external ABox into the knowledge graph:
  - *Entity resolution*, i.e., finding and linking duplicate instances
  - *Data fusion*, i.e., resolving property value conflicts after entity resolution

In this chapter, we will explain the knowledge enrichment process to improve the completeness of a knowledge graph. First, we will introduce the identification of new data and knowledge sources and potential ways to tackle it. Then, we will briefly cover the data lifting task as we have already covered it in the knowledge creation chapter (Chap. 18). Then we will focus on the integration of TBoxes, aka schema alignment, and the integration of ABox via new equality, instance, and property value assertions, including some strategies for fusion. Each step will be explained with the help of a running example based on the German Tourism Knowledge Graph (Fig. 22.1), where the knowledge graph is enriched with equality

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_22

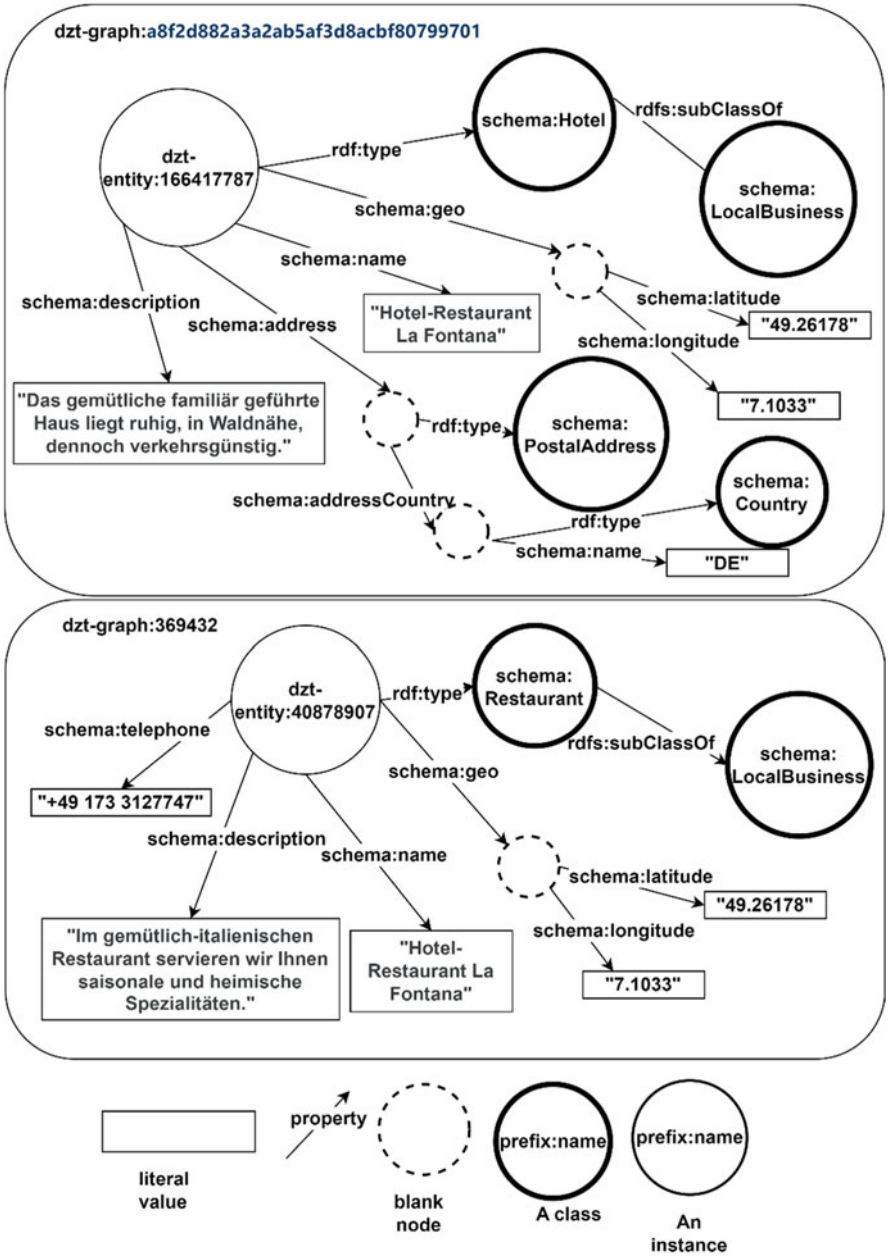


Fig. 22.1 An excerpt from the German Tourism Knowledge Graph is a running example to demonstrate knowledge enrichment

assertions across named graphs within the knowledge graph. We also use this example to give a larger illustration of the enrichment process before we conclude this chapter with a summary.

22.1 Identification of Additional Data and Knowledge Sources

Enrichment typically starts with the identification of new data sources. These sources could be already knowledge graphs, but could also be other unstructured, semi-structured, or structured sources, such as text; images; CSV, XML, and JSON documents; relational databases; and many more.

Automating the task of finding new knowledge sources is not very straightforward. Open sources can be semi-automatically discovered. Discovery queries can be used to identify if they cover the needs of our domain. Similarly, machine-readable descriptions of sources can be used, but unfortunately, such descriptions are rarely available. Many useful sources are proprietary, which require legal/commercial agreements between parties. This makes it much more challenging to do the discovery and access in an automated way.

The strategies for the identification of different sources may vary based on the domain. For instance, in tourism, there are common product and service aggregators that provide vast amount of data from a single point. They are, however, typically proprietary, and it may be challenging to access their data. Individual service providers can also be used, but there may be scalability issues as they are likely to be distributed. Moreover, there are open sources like Wikidata and DBpedia that contain cross-domain knowledge. These sources can be partially automatically discovered and accessed, for instance, via the discovery queries mentioned above.

22.2 Data Lifting

Not every external source comes in the form of a knowledge graph. Data providers use different formats and data models, such as spreadsheets, relational databases, and CSV, XML, and JSON files. These sources must first be "lifted," which describes the need of mapping their data syntax to the format of our knowledge graph.

The mapping can be done via dedicated wrappers programmatically. Such an approach may initially seem attractive as it gives the developer the power of a general-purpose programming language. However, it does not scale due to very low reusability and portability (see Chap. 18). The ideal way to do such mappings is to use declarative mapping languages. This way, the mapping engines can be developed generically, and mappings become more reusable. For example, the RDF Mapping Language (RML) (Dimou et al. 2014) can be used for this purpose

as it provides mechanisms to map various (semi-)structured sources such as relational databases and CSV, XML, and JSON files. There are also several format-specific tools and languages, such as Mapping Master (O'Connor et al. 2010) and XLWrap (Langegger and Wöß 2009) for spreadsheets, XMLtoRDF (Van Deursen et al. 2008) and Tripliser[1] for XML data, and R2RML[2] for relational databases.[3]

22.3 TBox Alignment

TBox alignment deals with the merging or alignment of the TBoxes of two knowledge graphs. TBox alignment happens between classes and properties, typically via building subsumption, equivalency, and domain/range relationships. Figure 22.2 shows the partial TBoxes of two knowledge graphs: the German Tourism Knowledge Graph and Wikidata.[4,5] One way to integrate these TBoxes is to define relationships between the classes of the two schemas. In Fig. 22.2, the *schema:Hotel* class and *wd:Q27686 (hotel)* are specified as equivalent classes with OWL. An equivalent class definition in OWL indicates that the extensions of the two classes are the same. Note that the alignment here could have also been done between classes that are one level higher in both hierarchies. Such an alignment, however, brings a stronger ontological commitment as it implies that all subclasses of each class are also subclasses of the other. Alternative to the equivalent class specification, two schemas can also be aligned with subclass relationships between classes. For example, the *schema:Hotel* type could be specified as the subclass of *wd:Q5056668 (lodging)*.

The properties across two TBoxes can also be aligned to integrate them. Figure 22.3 demonstrates two ways of doing this. In Fig. 22.3a, the property *schema:addressCountry* is defined as equivalent properties with *wd:P17 (country)* property. Consequently, the equivalency indicates that these two properties must have the same extension[6] (i.e., the sets defined by the properties contain the same subject and object pairs)**.** Another way of defining such an alignment is shown in Fig. 22.3b. There, the integration happens via a range definition, which extends[7] the range of the *schema:addressCountry* property with the *wd:Q6256 (country)* class, which

[1] https://daverog.github.io/tripliser/

[2] https://www.w3.org/TR/r2rml/

[3] See Chap. 18 for more tools that can be used for data lifting.

[4] Wikidata has a more complex model than RDF, but for more practical purposes, these primitives can be mapped to RDF. This also allows Wikidata to be published as an RDF dataset. See here https://www.wikidata.org/wiki/Wikidata:Relation_between_properties_in_RDF_and_in_Wikidata and here https://www.wikidata.org/wiki/Wikidata:RDF.

[5] *wd* is used as a prefix for the Wikidata namespace.

[6] https://www.w3.org/TR/owl-ref/#equivalentProperty-def

[7] The range is "extended" due to the disjunctive nature of range definitions in schema.org. With RDFS range definitions, the range would be strictly speaking "restricted."

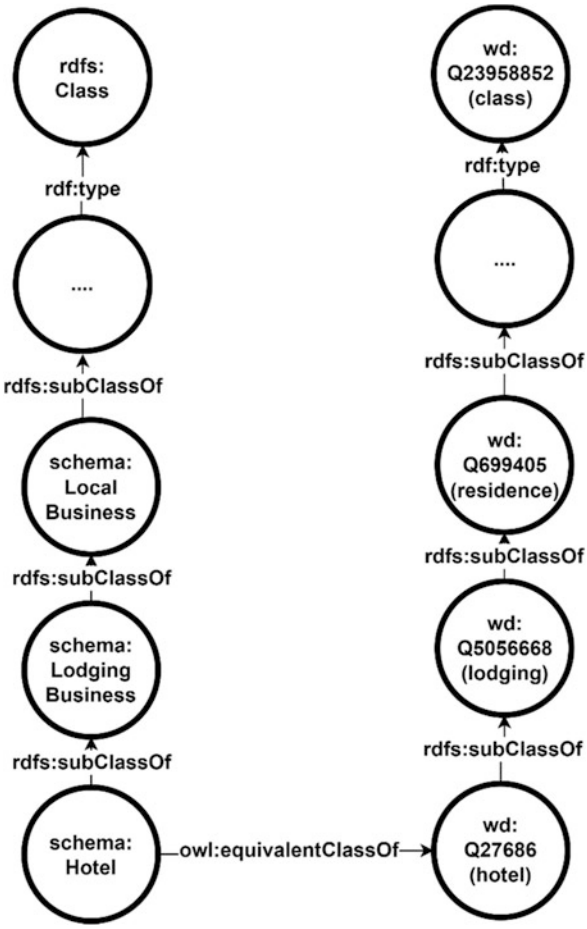


Fig. 22.2 Class alignment between TBoxes of the German Tourism Knowledge Graph (left) and Wikidata (right)

indicates that the instances of *wd:Q6256* can also be used as values of *schema:addressCountry*. Similarly, in Fig. 22.3c, the alignment happens via the domain definition, which extends the domain of the *schema:addressCountry* property with the *wd: Q98929991 (place)* class, which according to schema.org documentation[8] indicates that the *schema:addressCountry* property can also be used on the instances of *wd:Q98929991 (place)* type.

TBox alignment, also known as ontology alignment and merging, is a well-established research topic with a wide range of proposed solutions,[9] such as:

[8] See also https://schema.org/docs/datamodel.html.

[9] See also Part I about ontologies and their alignments and Sect. 18.1.4.

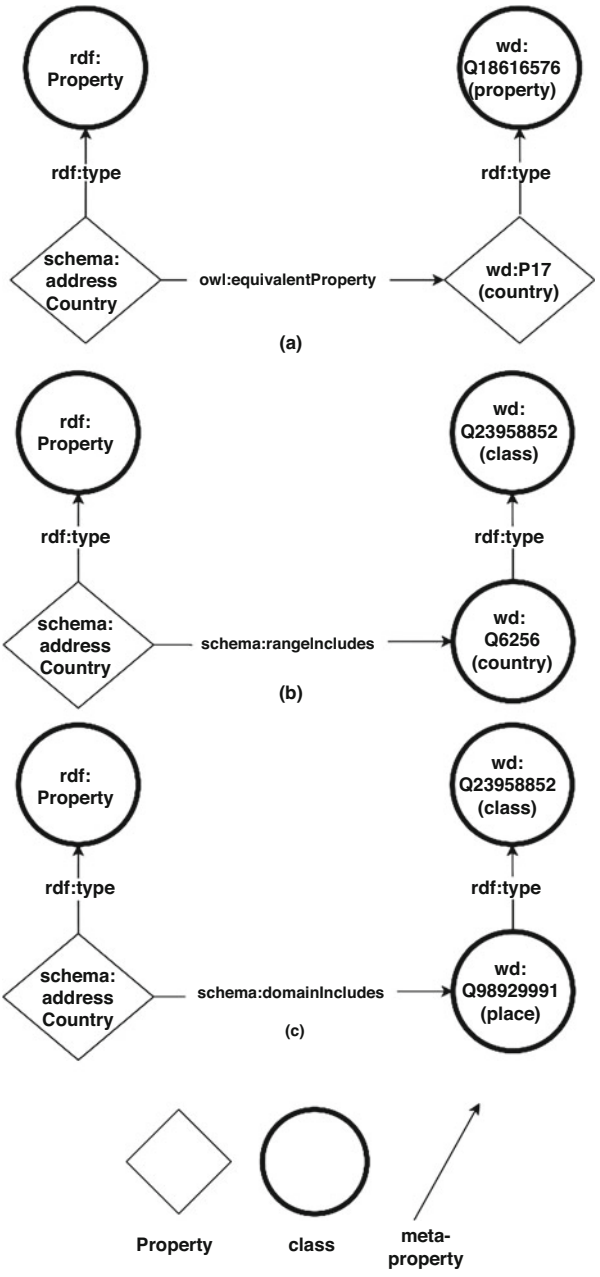


Fig. 22.3 TBox alignment via properties

- Deep Ontology MatchEr (DOME) (Hertling and Paulheim 2019)
- POMAP++ (Laadhar et al. 2018)
- Holontology (Roussille et al. 2018)
- ALIN (da Silva et al. 2020)
- LogMap (Jiménez-Ruiz and Grau 2011)
- ALOD2Vec (Portisch and Paulheim 2021)
- FCAMapX (Chen and Zhang 2018)
- KEPLER (Kachroudi et al. 2017)

As an example, we briefly introduce *DOME* and what TBox alignment with an approach like DOME can look like. DOME (Deep Ontology MatchEr) performs schema aligning by using the textual descriptions of resources taken from the literal-valued properties describing classes, properties, and individuals (instances) in an ontology. The process model of DOME for aligning types and properties of two ontologies consists of the following steps:

1. *String Matching* – initial matching of types, properties, and individuals in the ontologies based on specific string property values. For example, matching *rdfs:label* property values are a good indication that two resources can be aligned.
2. *Confidence Adjustment* – assigning new confidence values based on the similarity calculated from the vectorized literal values describing the resources.
3. *Instance-Based Type Alignment* – the types of the individuals that were matched with string-based methods are matched with a certain confidence value.
4. *Type Filter* – the alignments are filtered out if the types of two matched resources are different.
5. *Cardinality Filter* – when there are multiple matches for a type or a property, only the one with the highest confidence value is selected.
6. *Final Alignment* – the final aligned types and properties are given as output.

Among these steps, the first three are rather complex. We will explain those in the following with an indicative example.

String Matching*:* The first step of DOME is matching all resources in an ontology (classes, properties, and instances) via their textual representations. As a starting point, it applies some preprocessing on the literal values of certain properties and matches them with an initial confidence value:

- Matching *rdfs:label* values in a mapping with static confidence of 1.0
- Matching *skos:prefLabel* values in a mapping with static confidence of 1.0
- Matching URI fragment (i.e., the part that comes after the hash character #)[10] values in a mapping with static confidence of 0.9
- Matching *skos:altLabel* values in a mapping with static confidence of 0.9

The preprocessing step involves splitting text into tokens, transforming it into lowercase, and removal of stop words. Figure 22.4 shows two types from two

[10] https://en.wikipedia.org/wiki/URI_fragment. URI fragments are used commonly for identifying resources within a namespace. The namespace URI is followed by a # character which is followed by the name of the resource.

O_1[1]	**nco:PostalAddress** a rdfs:Class ; rdfs:label "PostalAddress"@en ; rdfs:comment """ A postal address. A class aggregating the various parts of a value for the 'ADR' property as defined in RFC 2426 Sec. 3.2.1. """@en .
O_2[2]	**schema:PostalAddress** a rdfs:Class ; rdfs:label "PostalAddress" ; rdfs:comment "The mailing address." .

Fig. 22.4 Two types from two ontologies. [1]https://www.semanticdesktop.org/ontologies/2007/03/22/nco/#PostalAddress; [2]https://schema.org/version/latest/schemaorg-current-https.ttl

O_1	Label= [postaladdress] Comment = [postal, address, class, aggregating, various, parts, value, adr, property, defined, rfc, 2426, sec, 3, 2, 1]
O_2	Label= [postaladdress] Comment= [mailing, address]

Fig. 22.5 The values of rdfs:label and rdfs:comment are preprocessed into ordered lists of tokens for both types

O_1	[postaladdress, postal, address, class, aggregating, various, parts, value , adr, property, defined, rfc, 2426, sec, 3, 2, 1]
O_2	[postaladdress, mailing, address]

Fig. 22.6 The types from both ontologies are represented as ordered lists of tokens based on their textual description

different ontologies (O_1 and O_2). These types are described with some properties, including two properties with string values, namely, *rdfs:label* and *rdfs:comment*.

For each type, the string literal values for *rdfs:label* and *rdfs:comment* properties can be preprocessed into an ordered list of tokens, as shown in Fig. 22.5.

For initial string matching, only the *rdfs:label* value is considered at this step. Since both types have the same *rdfs:label* value, the two types are aligned with an initial confidence value of 1.0.

Confidence Adjustment: After the initial matching, the confidence values are adjusted with the help of the vector representations of the textual descriptions of each resource. For this, for each already mapped resource, the triples where that resource is in the subject position are taken. Then, all property values among those triples with the datatype *xsd:string* or *xsd:langString* are preprocessed in the same fashion as the String Matching step and concatenated into a single ordered list of tokens per type. Note that here not only special properties like *rdfs:label* but all properties with string literal values are considered in the final list of tokens representation. Figure 22.6 shows the concatenated array of tokens representing each type.

In the next stage, these arrays of tokens are converted into vectors to obtain numerical representations for both types, which are then used for calculating a new

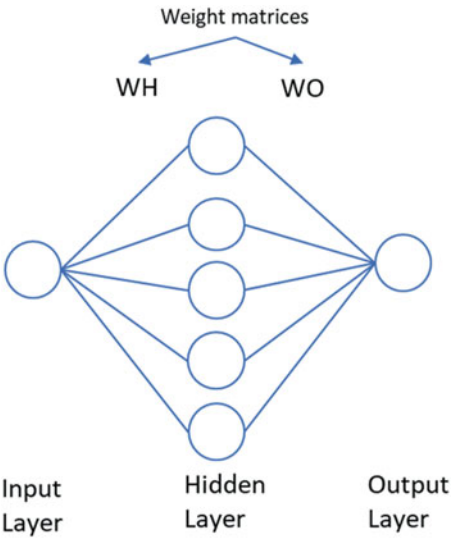


Fig. 22.7 A neural network with one hidden layer that can be used for vectorization

similarity score. A vectorization approach for this purpose is called doc2vec (Le and Mikolov 2014). *doc2vec* is an approach that converts the text of different length to fixed-length numerical vectors while considering their context (i.e., the words around a given word for a document).

The "document vectors" produced by the doc2Vec approach are a side product of a neural network with one hidden layer for a prediction task.[11] Figure 22.7 shows an illustration of a neural network with one hidden layer. *WH* and *WO* represent the weight vector between the input layer and hidden layer and hidden layer and output layer, respectively.

Assume that the task is "given a certain number of words in a document, predict the next word." For this task, the neural network is trained with this document iteratively, and this is done for each document. In a neural network, between each layer, there are weight matrices that transfer values from one layer to another. The weight values are initially assigned randomly and then adjusted after each iteration of training. What we use as document vectors are represented by the weight matrix between the input layer and the hidden layer (*WH*). For each document, the values from this weight matrix are seen as a vector representation of that document as those are the weights that lead to the most accurate prediction of the sequence of words in that document.

When we train the neural network in Fig. 22.7 with the ordered list of tokens in Fig. 22.6, we obtain two vectors in *WH*. The following are plausible vectors after such training[12]:

[11] Here, we show a quite simplified depiction of a vectorization approach aligned with the doc2Vec approach. More details of how doc2Vec can be implemented can be found in https://shuzhanfan.github.io/2018/08/understanding-word2vec-and-doc2vec/.

[12] The exact numbers may be different in each trained model as the initial weights are randomly assigned.

$$V_1 = [-0.2530334,\ 0.00625943,\ -0.20948619,\ -0.21726415,\ 0.0771472]$$

which represents the vector for O_1 and

$$V_2 = [-0.25462207,\ 0.00470902,\ -0.21381071,\ -0.21914613,\ 0.08078393]$$

which represents the vector for O_2. Note that the size of the vectors is fixed because it only depends on the size of the hidden layer. In this case, the hidden layer has five neurons; therefore, the vectors also have a size of five.

The cosine similarity[13] between V_1 and V_2 is used as a confidence value. It is calculated as the cosine value of the angle between two vectors. Cosine similarity is a popular measure in information retrieval for matching texts. Intuitively, we can say that the smaller the angle gets, the bigger the cosine value will be; therefore, it would indicate a higher similarity (Han et al. 2012). Figure 22.8 shows the cosine values for various degrees of angles.

Instance-Based Type Alignment: At this stage, DOME has already produced an alignment between instances, classes, and properties based on the previous steps. To align even more classes that may not be textually similar, the alignments between instances are used. The heuristic is "if two instances are aligned, the types of those instances may have a similar set of instances." The calculation is done via similarity metrics for sets derived from the Dice coefficient.[14] Given two sets, X and Y, the Dice coefficient is calculated via

$$\frac{\mathbf{2}|\boldsymbol{X}\cap\boldsymbol{Y}|}{|\boldsymbol{X}|+|\boldsymbol{Y}|}$$

which intuitively calculates how well two sets overlap. In the context of ontology alignment, the calculation returns a similarity score between two classes (sets of instances) based on the shared number of instances between those classes.

22.4 ABox Integration

ABox integration consists of two main tasks:

- *Entity resolution* – the alignment of instances by adding new statements that link them with each other
- *Data fusion* – fusing the property value assertions and resolving any conflicts

In the following, we will explain these two in detail.[15]

[13] https://en.wikipedia.org/wiki/Cosine_similarity

[14] https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient

[15] This section is partially based on Fensel et al. (2020).

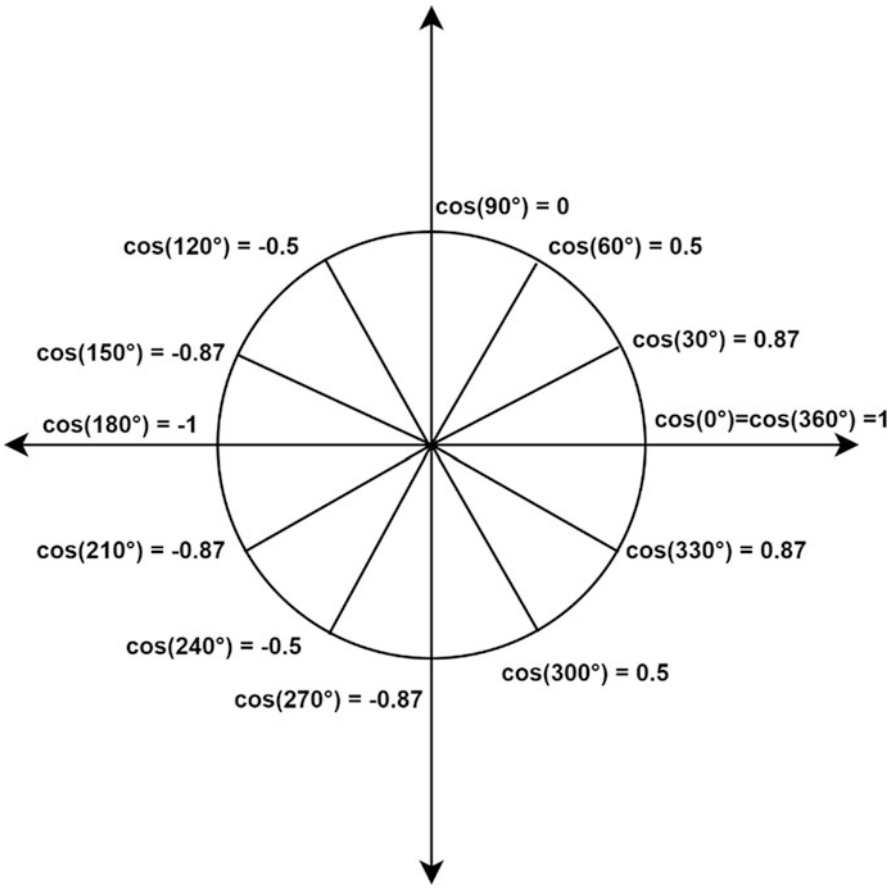


Fig. 22.8 Cosine values for different angles

22.4.1 Entity Resolution

The instances in a knowledge graph can be aligned in two ways, either via identifying duplicate instances, that is, the instances describing the same object in the domain, or via associating two arbitrary instances that are somewhat related with a property value assertion (e.g., two places can be linked via a containment relationship; POIs and gas stations nearby can be linked with a property). The latter scenario typically requires domain-specific rules.[16]

Duplicate detection is an important challenge in many fields that even go beyond computer science, and not surprisingly, the task itself has many names (see Fig. 22.9). This indicates that this task wrestles with a genuine and complex

[16] See Elmagarmid et al. (2007) for a survey of duplicate detection methods.

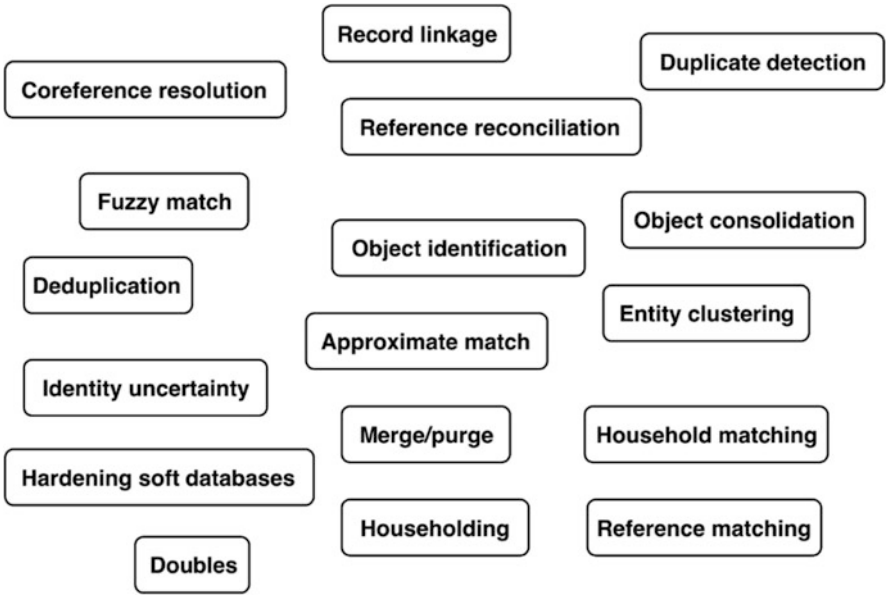


Fig. 22.9 The many names for the duplicate detection task (Getoor and Machanavajjhala 2012)

problem. Before we start with a rather generic process of duplicate detection, we assume that the TBox alignment is already completed, at least across the relevant schema for the instances that are subject to duplicate detection.

From a naïve starting point, a duplicate detection algorithm must compare all instances in a knowledge graph with all instances in another knowledge graph.[17] Assuming we have two knowledge graphs with M and N instances, respectively, we can say that without any heuristics, we have $M * N$ comparisons to make which is usually quite a large number. However, from a more practical point of view, this becomes quite infeasible, as knowledge graphs are ABox heavy and the number comparisons can easily exceed large magnitudes.

Several preprocessing techniques are frequently utilized to speed up the process. These are, for example, indexing for trading off space with time, blocking to eliminate obvious non-duplicates in advance, and parallel processing utilizing multiple-core CPUs. The entity resolution task involves a comparison of property value assertions on pairs of instances to decide whether two instances refer to the same entity. Therefore, the process model of duplicate detection (Fig. 22.10) is roughly the same across most implementations, although the specific implementation of each step may vary. In the following, we will explain each step in this process model.

[17] These two knowledge graphs can also be the same knowledge graph, if we are resolving the duplicate instances within a knowledge graph.

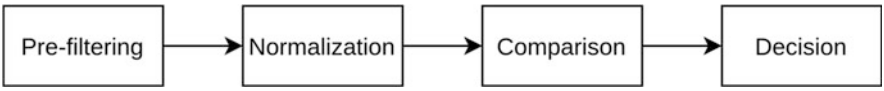


Fig. 22.10 The process model for duplicate detection

Pre-filtering: Pre-filtering techniques perform a very loose similarity computation between different instances to filter obvious "non-duplicates" from further inspection. An effective pre-filtering technique is *blocking*, where instances are grouped according to certain criteria. The blocking prevents instances that do not belong to the same "group" from being compared, eliminating unnecessary comparisons. Although blocking is effective, the *grouping strategy* carries the utmost importance to achieve good results in terms of recall and precision. We can give two example grouping methods for blocking: the type hierarchy and blocking keys (Papadakis et al. 2020; Vogel and Naumann 2012).

- The *type hierarchy* method is a simple grouping technique that allows the comparison of instances that share a common type or at least belong to a common type hierarchy. For example, two instances presented in Fig. 22.1 in German Tourism Knowledge Graph may end up in the same group as they are both instances of *schema:Hotel*.[18] This criterion can be relaxed by allowing the comparison of all *schema:LodgingBusiness* instances, which include the instances of all of its subtypes. The type hierarchy heuristic prevents, for example, the comparison of *schema:Hotel* instances with *schema:Event* instances, as they do not share any supertypes until *schema:Thing* in the class hierarchy.
- The *blocking keys* method uses the heuristic that the instances may be grouped based on certain property values that are more likely to identify instances uniquely. These properties can be decided either based on formal specifications such as functional properties or via domain knowledge. For example, the postal code and country can be used together as a criterion for further grouping the *schema:Hotel* instances. It is pretty unlikely that two hotels in different countries and with different postal codes are duplicates.

Normalization: During the duplicate detection process, comparisons are typically made between literal values. This requires a certain level of normalization as the next step. Through normalization, property values can be represented more uniformly to streamline the comparison process. Many approaches provide the functionality to transform values like removing capitalization, removing stop words, enforcing a fixed decimal length for numbers, and many more.

Comparison: The comparison step calculates a similarity score for two instances based on the values of certain properties in those instances. Comparing two properties results in a similarity value $\in [0, 1]$. Different properties might require different comparison functions. For example, properties with Boolean values simply produce

[18] An integrated TBox becomes quite useful for this grouping strategy.

a value of either 0 or 1. String-valued properties can be compared via string similarity measurement techniques such as the Levenshtein distance, also known as edit distance. The Levenshtein distance calculates the number of edits required to make one string the same as the other one. We normalize this distance to the [0,1] range to obtain a similarity score with the following formula:

$$1 - \frac{d}{\max(|s_1|, |s_2|)}$$

where d is the edit distance and $|s_1|$ *and* $|s_2|$ are the lengths of the first and second strings, respectively. The result will be closer to 1 if the edit distance is smaller.

More complex properties like geocoordinates may be compared with Euclidean distance.[19] Given a pair of coordinates (p_1, p_2) and (q_1, q_2), the Euclidean distance between these coordinates is calculated as $\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$. Although Euclidean distance is typically enough for comparison, a more accurate distance calculation can be done with the Haversine distance,[20] given the spherical shape of the earth. The Haversine distance measures the shortest distance between two coordinates on a spherical surface.

The values of multiple properties can also be combined before comparison, such as in the case of the postal address. An important challenge for the comparison step is multi-valued properties. How do we compare a list of values? There are several ways to handle multi-valued properties, for example:

- *Aggregation-first-similarity-second strategy:* This strategy first aggregates the values of a property on each instance (e.g., via summation or average) and then compares them. For example, if we have a geo-shape defined by multiple geocoordinate values, we can first aggregate values per instance (e.g., finding the central point of the shape) and then calculate the Euclidean distance between the central points aggregated on each instance.
- *Similarity-first-aggregation-second strategy:* This strategy first calculates the similarity between each individual value and then calculates their aggregation (e.g., mean) as the similarity value. For example, if we want to use the opening hours of an establishment as part of the duplicate detection process, we can compare the opening hours each day of the week on each instance via a string similarity measure and then aggregate the similarity scores to obtain a single score for the opening hours property.

Decision: After the comparison step, we obtain a similarity value for each pair of properties in two instances. These similarity scores must be somehow translated into a decision. For obtaining an aggregated similarity score for two instances, various approaches can be adopted. *Similarity aggregation* (Tran et al. 2011) simply

[19] https://en.wikipedia.org/wiki/Euclidean_distance

[20] For the calculation of Haversine distance, we refer readers to https://en.wikipedia.org/wiki/Haversine_formula.

calculates the mean of all similarity scores. *Weighted similarity aggregation* (Tran et al. 2011) does the same, except each property has assigned a weight. These two aggregation methods are simple; however, they do not consider the positive or negative "impact" of each property that is involved in calculating the similarity scores based on *prior* knowledge. For a calculation considering such an impact, Bayes' theorem can be used (Elmagarmid et al. 2007). Bayes' theorem defines the probability of an event based on prior knowledge about the conditions affecting that event. Given two events, A and B, their probabilities of happening are $P(A)$ and P(B), respectively. $P(A \mid B)$ is the conditional probability of observing A given that B is observed, and $P(B \mid A)$ is the vice versa. Bayes' theorem is written as

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

In the context of duplicate detection, we can consider A as the event of two instances being duplicates and B as the event of a certain property on two instances having the same value. Naturally, in realistic scenarios, multiple properties are involved at the same time.

Note that these probabilities are not arbitrarily assigned. We use prior domain knowledge to estimate them. In countries like Austria and Germany, it is very common for hotels to have similar names. You can just search for "Hotel Post" and see how many hotels you will get. Therefore, the probability of two hotels being duplicates when they have a similar name is closer to 0.5, which would reduce the impact of the *name* property. Contrarily, having the same address has a more significant indication that the two hotels are the same. Therefore, the probability of two hotels being duplicates if they have the same address can be assigned a high value like 0.9. Alternatively, the results of the previously completed duplicate detection task can also be analyzed to obtain these prior probabilities.

There is a wide landscape of tools for duplicate detection, including but not limited to:

- ADEL (Plu et al. 2017)
- Dedupe[21]
- Duke[22] (Garshol and Borge 2013)
- Legato (Achichi et al. 2017)[23]
- LIMES (Ngomo et al. 2021)[24]
- Serimi (Araujo et al. 2011)
- Silk (Volz et al. 2009)[25]

[21] https://github.com/dedupeio/dedupe

[22] https://github.com/larsga/Duke

[23] https://github.com/DOREMUS-ANR/legato

[24] http://aksw.org/Projects/LIMES.html

[25] http://silkframework.org/

Duke (Garshol and Borge 2013) is an entity resolution engine that implements the aforementioned process model. It uses a Lucene index for fast access to the instances and blocking techniques like blocking keys for pre-filtering. The configuration contains information about which properties are relevant for entity resolution and how their values can be normalized (e.g., lowercase transformation, cleaning stop words, etc.). The users can also define specific similarity functions for each value for the comparison step (e.g., Levenshtein distance, Euclidean distance, etc.). The properties used in the comparison can be weighted, and individual similarity scores are combined with the Bayesian method while also considering the impact of certain properties being "not the same" on two instances being duplicates.[26]

22.4.2 Data Fusion

Now that we know which representations refer to the same object of discourse, we can merge these representations. The duplicates are typically linked with the *owl: sameAs* property. As the next step, we can merge the property value assertions from the linked duplicate instances. Integrating new data sources can cause inconsistencies like functional properties having multiple distinct values or newly introduced property value assertions violating existing constraints (e.g., postal code from the new source does not fit to the defined pattern in a SHACL shape). The solution is to apply a knowledge cleaning process.

While resolving errors caused during data fusion, detection of syntactic errors, and violation of formal specification are typically straightforward, other situations where all values of a property are formally correct can be tricky to resolve. There are two main assumptions, namely, single-truth and multi-truth (Azzalini et al. 2019).

- In the *single-truth assumption*, we consider only one property value as the correct value for a property. There are several strategies that can be adopted for selecting a "truth" for a property value. We can keep the most frequently appearing value, use aggregations (average, max, min) for numerical values, define a confidence threshold based on the characteristics of the source (e.g., trustworthiness, recency), and adopt crowdsourcing.
- For the cases a property allows *multiple values*, Bayesian theory can be used to pick the most probable values (Dong et al. 2009; Wang et al. 2015; Wu and Marian 2007). Naturally, the strategies for single-truth assumption can iteratively also be used to support the decision for multi-truth situations.

For various fusion strategies, an important heuristic to use is the quality of the external source selected, particularly in dimensions like trustworthiness and reliability (Li et al. 2016), as well as recency. The measurement of such dimensions can be largely manual but can also be automated to some extent. For example, knowledge

[26] https://www.garshol.priv.no/blog/217.html

graphs in LOD that have a large number of incoming links can be seen as trustworthy sources.[27] Platforms like LOD Cloud[28] provide machine-processable metadata about every linked open dataset. This metadata includes the information on which datasets are linked together. This information can be used to identify the number of incoming and outgoing links for each dataset.

There are several tools for the data fusion task. These are including but are not limited to:

- FAGI[29] (Giannopoulos et al. 2014)
- FuSem (Bleiholder et al. 2007)
- HumMer (Bilke et al. 2005)
- KnoFuss[30] (Nikolov et al. 2008)
- ODCleanStore (Knap et al. 2012)
- Sieve[31] (Mendes et al. 2012)

Among these tools, Sieve is an example of a data fusion tool that benefits from quality assessment results to make decisions about which property value to keep during cleaning after fusion. Sieve fusion functions are basically of two types (see also Fensel et al. (2020)):

- *Filter functions (deciding strategies)* remove values according to some quality metric:
  - *Filter* removes values for which the input quality assessment metric is below a given threshold.
  - *KeepSingleValueByQualityScore* keeps only the value with the highest quality assessment.
  - *First, Last, Random* takes the first, the last, or the value at some random position.
  - *PickMostFrequent* selects the value that appears more frequently in the list of conflicting values.
- *Transform functions (mediating strategies)* operate over each value in the input, generating a new list of values built from the initially provided ones:
  - *Average* takes the average of all input values for a given numeric property.
  - *Max* chooses the maximum of all input values for a given numeric property.
  - *Min* chooses the minimum of all input values for a given numeric property.

Among these fusion functions, the filter functions provide an important feature that benefits from the quality assessment of integrated sources. The intuition is "if I

[27] See Kleinberg (1999) and Borodin et al. (2005) for an example approach for finding out which Web pages are authoritative.

[28] https://lod-cloud.net/

[29] https://github.com/GeoKnow/FAGI-gis

[30] http://technologies.kmi.open.ac.uk/knofuss/

[31] http://sieve.wbsg.de/

have two feasible values from two sources, I pick the one from a higher quality source in terms of some quality dimensions." Sieve uses several assessment dimensions and metrics to prioritize sources. For example, *recency* is a dimension that ranks more recently updated graphs higher. The *trustworthiness* dimension prioritizes the most trustworthy sources. The sources can be ranked manually by a domain expert, or metrics-based numbers of incoming links can also be used, similar to the PageRank algorithm (see Chap. 3).

22.5 Illustration: Enriching the German Tourism Knowledge Graph

In this section, we provide a larger illustration of the knowledge enrichment process. More concretely, we will detect duplicate instances across two named graphs and link them with the *owl:sameAs* statements. Afterward, we will apply the data fusion process to "merge" two instances into a single representation.

22.5.1 Duplicate Detection

Figure 22.1 contains two named graphs from the German Tourism Knowledge Graph. The first step of the duplicate detection process is to decide whether the instances in the two graphs are "potentially" duplicates. When the blocking strategy based on the type hierarchy is applied, it becomes clear that these two instances are good candidates for a similarity comparison, as they share the same *schema:Hotel* type. After a normalization step where, for example, whitespaces in literal values are removed and all words converted to lowercase, the comparison can start. The crucial point here is to pick the right properties that typically identify instances well. In this example, we pick *schema:name*, *schema:description*, and *schema:geocoordinates* for comparison. Then, these properties are applied to both instances. The comparison of these properties can be made via string similarity measurements for *schema:name* and *schema:description* and with Haversine distance for *schema:geocoordinates*.

Let us assume we use Levenshtein distance to measure the similarity between name and description values, respectively. The result of this calculation will be closer to 1, if the edit distance is small, which means the strings are similar. As the *schema:name* values are the same for both instances, they have a similarity score of 1. The edit distance between the two *schema:description* values is 67, and the longer string has 96 characters. Therefore, we can calculate a similarity score of

$$1 - \frac{67}{96} = 0.3.$$

The distance between two geocoordinates can be calculated with the Haversine distance. The normalization of the Haversine distance to [0, 1] can be done by deciding on a maximum distance of such as 100 km. This means any distance longer than 100 km produces a similarity value of 0, while a distance of 0 produces a similarity value of 1. The normalization formula could be the following:

$$\max\left(\left(1-\frac{d}{100}\right),0\right),$$

where d is the Haversine distance between two geocoordinates. Since the geocoordinates on both instances are identical, we calculate a similarity score of 1.

After calculating individual similarity scores for all selected properties for comparison, we need to aggregate these scores into a single similarity value for two instances. For the sake of simplicity, we select the weighted similarity aggregation method as described in Sect. 22.4.1. An important decision here is to decide on the weights. With our domain knowledge, we can conclude that geocoordinates are better indicators of whether two places are the same than name and description, description particularly being the lowest. Assume we have weights 0.6 for geocoordinates, 0.3 for name, and 0.1 for description.[32] This calculates to

$$0.6 * 1 + 0.3 * 1 + 0.1 * 0.3 = 0.93.$$

Based on this high score, we can conclude that these two instances are duplicates, assuming that the threshold we determined is below 0.93.

An interesting approach would be to not only consider the positive contribution of a property for similarity but also the negative contribution. For example, having a high similarity score for the Social Security number of two instances can indicate a high probability (significantly above 0.5) of them referring to the same person, whereas having a low similarity score for that property can indicate a low probability (significantly below 0.5) of them referring to the same person. These probabilities obtained from different properties can then be combined with the Bayesian method. As mentioned before, the Duke tool works with this principle.

22.5.2 Data Fusion

Luckily, for data fusion, we do not have a complicated situation. The only conflicting values we have are the values of the description properties. Due to the RDF data model, property values that exist only in one of the instances can be fused seamlessly.

[32] Duplicate detection of point of interests may be much more challenging. See Athanasiou et al. (2019) for some major issues that can occur in real-world scenarios.

The description property on each instance has different values. We have multiple options here to decide which ones we keep. The first option is to keep them both, as there is no known restriction on the cardinality of the description property. The second option is to decide on a single truth. Since there are no formal restrictions violated by any of the values, we need to rely on a strategy that considers other factors, like the quality of the sources from which the instance values are coming. If you remember from knowledge hosting, the named graphs contain provenance information, including when the knowledge is generated. We can pick the value that is more recently generated for the description since both instances come from the same source. It is more likely that the more recently generated value is the more desired one for applications to use.

22.6 Summary

Knowledge enrichment is a hard and important task. It aims to improve the completeness of a knowledge graph. Improving completeness means reducing the number of missing assertions. The most important consideration for knowledge enrichment is to decide "what is missing." This contextual information is typically provided via two paths, namely, completeness regarding a formal specification and completeness regarding a specific use case. The collection of this information is typically done at the assessment step in the lifecycle. Once the need for enrichment is identified, the process consists of the following steps: identification of new sources, lifting of new sources if needed, integrating TBox, and integrating ABox.

The first step is typically done mostly manually, although some automation is possible for open knowledge sources. Naturally, not every external source is a knowledge graph, but they could be in various formats such as spreadsheets, CSV, JSON, and so on. The data lifting step maps their data to the RDF data model. For this task, knowledge creation methods can be adopted. Afterward, TBox alignment is conducted. This step aligns the schema of two sources, and it is already a very complicated but well-researched task on its own. Finally, ABox integration is the alignment of instances between knowledge graphs and itself consists of two tasks: namely, linking instances and fusing them into a single representation. Instances can be linked via identity or association links with each other.

In this chapter, we focused on identity links, also known as duplicate detection. The main idea behind duplicate detection is to find instances that are so alike that they can be considered duplicates. The comparison is made at the property value level. We discussed various similarity measurements for different types of property values. After we calculate individual similarity scores, we aggregate them and decide whether it has reached an instance similarity score high enough to consider them duplicates.

Once two duplicates are linked, the property value assertions describing them can be merged. This process is called data fusion, where two instances are merged into a single representation and any conflicts occurring are solved via knowledge cleaning.

As we have seen in the knowledge cleaning chapter, it is rather straightforward to identify syntactic and some formal errors; however, identifying domain-related semantic errors and correcting them might be tricky. The main challenge comes from picking a set of values as *truth* among a larger set of values of a property. We have covered many strategies for this, mainly involving aggregation and additional heuristics regarding the quality of external sources where the property values are coming from.

Knowledge enrichment is the last step of the knowledge curation process we present in this book. In practice, there is an interplay between cleaning and enrichment, as you may have already realized. During cleaning, some errors are corrected by adding new statements, and during enrichment, cleaning may be necessary when data fusion is conducted. In the last major chapter of Part III, we will talk about how curated knowledge graphs can be deployed.

References

Achichi M, Bellahsene Z, Todorov K (2017) Legato: Results for OAEI 2017. In: Proceedings of the 12th International Workshop on Ontology Matching (OM2017) co-located with the 16th International Semantic Web Conference (ISWC2017). CEUR Workshop Proceedings, Vienna, Austria, October 21, vol 2032, pp 146–152

Araujo S, Hidders J, Schwabe D, De Vries AP (2011) SERIMI-resource description similarity, RDF instance matching and interlinking. In: Proceedings of the 12th International Workshop on Ontology Matching (OM2017) co-located with the 16th International Semantic Web Conference (ISWC2017). CEUR Workshop Proceedings, vol 2032, Vienna, Austria, October 21

Athanasiou S, Alexakis M, Giannopoulos G, Karagiannakis N, Kouvaras Y, Mitropoulos P, Patroumpas K, Skoutas D (2019) SLIPO: large-scale data integration for points of interest. In: Proceedings of the 22nd International Conference of Extending Database Technology (EDBT), Lisbon, Portugal, March 26–29, OpenProceedings.org, pp 574–577

Azzalini F, Piantella D, Tanca L, et al (2019) Data fusion with source authority and multiple truth. In: Proceedings of the 27th Italian Symposium on Advanced Database Systems, Castiglione della Pescaia (Grosseto), Italy, June 16–19

Bilke A, Bleiholder J, Bohm C, Draba K, Naumann F, Weis M (2005) Automatic data fusion with hummer. In: VLDB'05: Proceedings of the 31st international conference on Very large data bases. ACM, pp 1251–1254

Bleiholder J, Naumann F (2009) Data fusion. ACM Comput Surv 41(1):1–41

Bleiholder J, Draba K, Naumann F (2007) FUSEM - exploring different semantics of data fusion. VLDB 7:1350–1353

Borodin A, Roberts GO, Rosenthal JS, Tsaparas P (2005) Link analysis ranking: algorithms, theory, and experiments. ACM Transactions on Internet Technology (TOIT) 5(1):231–297

Chen G, Zhang S (2018) FCAMapX results for OAEI 2018. In: OM@ISWC, CEUR-WS.org, CEUR Workshop Proceedings, vol 2288, pp 160–166

da Silva J, Revoredo K, Baiao FA, Euzenat J (2020) Alin: improving interactive ontology matching by interactively revising mapping suggestions. Knowl Eng Rev 35:e1

Dimou A, Vander Sande M, Colpaert P, Verborgh R, Mannens E, Van de Walle R (2014) RML: a generic language for integrated RDF mappings of heterogeneous data. LDOW:1184

Dong XL, Berti-Èquille L, Srivastava D (2009) Integrating conflicting data: The role of source dependence. Proc VLDB Endow 2(1):550–561

Elmagarmid AK, Ipeirotis PG, Verykios VS (2007) Duplicate record detection: A survey. IEEE Trans Knowl Data Eng 19(1):1–16. http://dblp.uni-trier.de/db/journals/tkde/tkde19.html#ElmagarmidIV0

Fensel D, Simsek U, Angele K, Huaman E, Kärle E, Panasiuk O, Toma I, Umbrich J, Wahler A (2020) Knowledge graphs. Springer

Fürber C, Hepp M (2011) SWIQA - a semantic web information quality assessment framework. In: Proceedings of the 19th European Conference on Information Systems (ECIS2011), Association for Information Systems (AIS eLibrary), Helsinki, Finland, June 9–11, p 76

Garshol LM, Borge A (2013) Hafslund Sesam – an archive on semantics. In: The Semantic Web: Semantics and Big Data: 10th International Conference, ESWC 2013, Montpellier, France, May 26–30, 2013. Proceedings 10, Springer, pp 578–592

Getoor L, Machanavajjhala A (2012) Entity resolution: Theory, practice & open challenges. Proc VLDB Endow 5(12):2018–2019. https://doi.org/10.14778/2367502.2367564. http://vldb.org/pvldb/vol5/p2018 lisegetoor vldb2012.pdf

Giannopoulos G, Skoutas D, Maroulis T, Karagiannakis N, Athanasiou S (2014) FAGI: A framework for fusing geospatial RDF data. In: OTM Conferences 2014, Amantea, Italy, October 27–31, Lecture Notes in Computer Science, vol LNCS 8841. Springer, pp 553–561

Han J, Kamber M, Pei J (2012) Getting to know your data. In: Data mining, 3rd edn. The Morgan Kaufmann Series in Data Management Systems, pp 39–82

Hertling S, Paulheim H (2019) DOME results for OAEI 2019. In: Proceedings of the 14th International Workshop on Ontology Matching, OM@ISWC 2019, Auckland, New Zealand, October 26, CEUR-WS.org, CEUR Workshop Proceedings, vol 2536, pp 123–130

Jiménez-Ruiz E, Grau BC (2011) LogMap: logic-based and scalable ontology matching. In: The Proceedings of 10th International Semantic Web Conference ISWC (1), Springer, Lecture Notes in Computer Science, vol 7031, pp 273–288

Kachroudi M, Diallo G, Yahia SB (2017) OAEI 2017 results of KEPLER. In: OM@ISWC, CEUR-WS.org, CEUR Workshop Proceedings, vol 2032, pp 138–145

Kleinberg JM (1999) Authoritative sources in a hyperlinked environment. J ACM 46:604–632

Knap T, Michelfeit J, Necasky M (2012) Linked open data aggregation: Conflict resolution and aggregate quality. In: COMPSAC Workshops, Izmir, Turkey, July 16–20, IEEE Computer Society, pp 106–111

Laadhar A, Ghozzi F, Megdiche I, Ravat F, Teste O, Gargouri F (2018) OAEI 2018 results of POMap++. In: Proceedings of the 13th International Workshop on Ontology Matching, OM@ISWC 2018, Monterey, CA, USA, October 8, CEUR-WS.org, CEUR Workshop Proceedings, vol 2288, pp 192–196

Langegger A, Wöß W (2009) XLWrap - querying and integrating arbitrary spreadsheets with SPARQL. In: Proceedings of the 8th International Semantic Web Conference (ISWC 2009), Springer, Lecture Notes in Computer Science, vol 5823, pp 359–374.

Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: International conference on machine learning, PMLR. ICML, pp 1188–1196

Li Y, Gao J, Meng C, Li Q, Su L, Zhao B, Fan W, Han J (2016) A survey on truth discovery. ACM SIGKDD Explor Newsl 17(2):1–16

Mendes PN, Mühleisen H, Bizer C (2012) Sieve: linked data quality assessment and fusion. In: In Proceedings of the 2nd International Workshop on Linked Web Data Management (LWDM 2012), in conjunction with the 15th International Conference on Extending Database Technology (EDBT2012): Workshops, ACM, Berlin, Germany, March 30, pp 116–123

Ngomo AN, Sherif MA, Georgala K, Hassan MM, Dreßler K, Lyko K, Obraczka D, Soru T (2021) LIMES: A framework for link discovery on the semantic web. Künstliche Intell 35(3):413–423

Nikolov A, Uren VS, Motta E, Roeck AND (2008) Integration of semantically annotated data by the KnoFuss architecture. In: In Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW2008): Practice and Patterns, Springer LNCS 5268, Acitrezza, Italy, September 29–October 2, Springer, Lecture Notes in Computer Science, vol 5268, pp 265–274

O'Connor MJ, Halaschek-Wiener C, Musen MA, et al. (2010) Mapping master: A flexible approach for mapping spreadsheets to owl. In: In Proceedings of the 9th International Semantic Web Conference (ISWC2010): Revised Selected Papers, Springer LNCS 6497, Shanghai, China, November 7–11, pp 194–208

Papadakis G, Skoutas D, Thanos E, Palpanas T (2020) A survey of blocking and filtering techniques for entity resolution. arXiv preprint arXiv:190506167

Plu J, Troncy R, Rizzo G (2017) ADEL: A generic method for indexing knowledge bases for entity linking. In: SemWebEval@ESWC, vol 769. Springer, Communications in Computer and Information Science, pp 49–55

Portisch J, Paulheim H (2021) Alod2vec matcher results for OAEI 2021. In: OM@ISWC, CEUR-WS.org, CEUR Workshop Proceedings, vol 3063, pp 117–123

Roussille P, Megdiche I, Teste O, Trojahn C (2018) Holontology: results of the 2018 OAEI evaluation campaign. In: Proceedings of 13th International Workshop on Ontology Matching co-located with the 17th International Semantic Web Conference (OM@ISWC 2018), Monterey, CA, United States, CEUR-WS.org, CEUR Workshop Proceedings, vol 2288, pp 167–172

Tran QV, Ichise R, Ho BQ (2011) Cluster-based similarity aggregation for ontology matching. Ontol Match 814

Van Deursen D, Poppe C, Martens G, Mannens E, Van de Walle R (2008) XML to RDF conversion: a generic approach. In: 2008 International conference on automated solutions for cross media content and multi-channel distribution. IEEE, pp 138–144

Vogel T, Naumann F (2012). Automatic blocking key selection for duplicate detection based on unigram combinations. In: Proceedings of the International Workshop on Quality in Databases (QDB)

Volz J, Bizer C, Gaedke M, Kobilarov G (2009) Discovering and maintaining links on the web of data. In: Proceedings of the 8th International Semantic Web Conference (ISWC2009), Chantilly, USA, October 25–29, Springer, Lecture Notes in Computer Science, vol LNCS 5823, pp 650–665

Wang X, Sheng QZ, Fang XS, Yao L, Xu X, Li X (2015) An integrated Bayesian approach for effective multi-truth discovery. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM'15), ACM, pp 493–502

Wu M, Marian A (2007) Corroborating answers from multiple web sources. In: Proceedings of the 10th International Workshop on the Web and Databases (WebDB2007), Beijing, China, June 15

Zaveri A, Rula A, Maurino A, Pietrobon R, Lehmann J, Auer S (2016) Quality assessment for linked data: a survey. Semant Web 7(1):63–93

Chapter 23
Tooling and Knowledge Deployment



Knowledge deployment in the knowledge graph life cycle is the last step, where the curated knowledge is deployed to be consumed by applications. From different perspectives, knowledge deployment can be categorized in different ways. For example, from a legal point of view, we can talk about open and proprietary knowledge; from an availability and freshness point of view, we can talk about static Resource Description Framework (RDF) dump files and live query interfaces. This chapter will cover knowledge deployment from a technical perspective, namely, highly distributed Web annotations and knowledge graphs (open or proprietary) that provide access to large integrated knowledge.[1]

The Web annotation approach focuses on making the content, data, and services on the Web machine understandable via semantic annotations. The most prominent example of this approach is the schema.org annotations that we have already covered in Part II. The advantage of this approach is that the technical hurdles are relatively low, and the effort required is split among basically millions of people who publish something as simple as a Web page on the Web. In many cases, popular content management systems (e.g., WordPress[2]) already provide some semantic annotations out of the box. The semantically annotated data then must be crawled and curated by applications.

Deployment as knowledge graphs requires a greater technical effort and infrastructure on the publisher side; however, it also provides easier access to a large amount of knowledge for the consumers. Additionally, the curation effort can already be taken on the knowledge graph, which makes the lives of consumers easier.

[1] There are also emerging approaches like *data spaces*, which enable a federated deployment of metadata, without strict constraints on the format and modeling of the data published by different service providers. See Chap. 25 for some details about them.

[2] https://wordpress.org/

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_23

In this chapter, first, we will introduce an example tool, a platform for the knowledge graph life cycle that can support both types of knowledge deployment. Afterward, we will present some potential challenges of knowledge deployment and an approach to go around those challenges, namely, an access and representation layer. Finally, we will provide a summary of knowledge deployment.

23.1 Tooling for the Knowledge Graph Life Cycle

There are many tools supporting different steps of the knowledge graph life cycle, many of which we already mentioned in the corresponding chapters in Part III. In this section, we will cover more holistic tools that provide a suite for creation, hosting, curation, and deployment.

There are not too many suites that approach the knowledge graph life cycle holistically. Some approaches provide pluggable frameworks, like Helio (Cimmino and Garcia-Castro 2023), which provides components for knowledge creation and hosting, and external curation tools can be plugged in via a SPARQL endpoint. We will go into detail about a platform called *Semantify.it*[3] (Kärle et al. 2017) that has been developed intensively in recent years. It is the offspring of industrial research cooperation between feratel AG, Onlim GmbH, and the University of Innsbruck.[4] As part of the exploitation agreement, the platform is free to use and is currently hosted and maintained by Onlim GmbH.[5] Semantify.it supports the knowledge graph life cycle to a large extent. In the remainder of the section, we will cover various modules of the tool for creation, hosting, assessment, and cleaning and an external tool for enrichment that can be used on the knowledge graphs developed with Semantify.it.

23.1.1 Knowledge Creation

In Chap. 18, we showed methodologies for knowledge creation. In this section, we show tool support from the Semantify.it platform for knowledge creation. The platform allows knowledge creation via different methods and stores it in so-called *collections*. Every registered organization can manage multiple collections of annotation on Semantify.it, for example, to be deployed on different Web sites or as a knowledge graph. In the following, we introduce the tool support for manual and mapping-based knowledge creation.

[3] https://semantify.it

[4] https://mindlab.ai

[5] The functionality explained in this section requires a login. Registration is free and quite straightforward from the homepage.

23.1.1.1 Manual Annotation Editor

The *manual annotation editor* (Fig. 23.1) enables users to create semantic annotations for low-volume and rather static data. The editor is generated based on a domain specification (see also Sect. 18.2.1), which provides significant convenience to the user. Only the properties of a type relevant to the domain are shown, and the input fields are generated based on the constraints defined on each property and their expected types (ranges). For example, if a property expects a date value, then a date picker is created in the interface.

A more compact version of the manual annotation editor is developed as a JavaScript library that can be integrated on arbitrary Web sites and as plugins for popular content management systems like WordPress and Typo3. The top side of Fig. 23.1 shows a compact editor for annotating events.

23.1.1.2 Mapper

Manual knowledge generation becomes infeasible as soon as the volume and the velocity of the data increase. In many cases, knowledge is generated based on existing nonsemantic (semi-)structured data sources. Therefore, for most knowledge graphs, some sort of mapping between the metadata of an arbitrary data source and an ontology is required. For the sake of reusability and understandability, declarative languages are typically used to create such mappings. We have covered some of these already in Chap. 18, with a particular focus on the RDF Mapping Language (RML).

Semantify.it offers RocketRML[6] (Şimşek et al. 2019), which allows the semi-automated integration of knowledge from heterogeneous data sources. Figure 23.2 shows the interface of the mapping module. It is quite straightforward as it only requires the Uniform Resource Locator (URL) of the source (e.g., a JavaScript Object Notation (JSON) file or a Web Application Programming Interface (API) endpoint) and the URL of the RML mapping file. Multiple sources and mapping files can be run in parallel. Moreover, cron jobs[7] can be configured to run the mappings periodically.

23.1.2 Knowledge Hosting

To support two different knowledge deployment approaches efficiently and effectively, Semantify.it uses a twofold hosting mechanism. The created semantic anno-

[6] https://github.com/semantifyit/RocketRML

[7] https://en.wikipedia.org/wiki/Cron

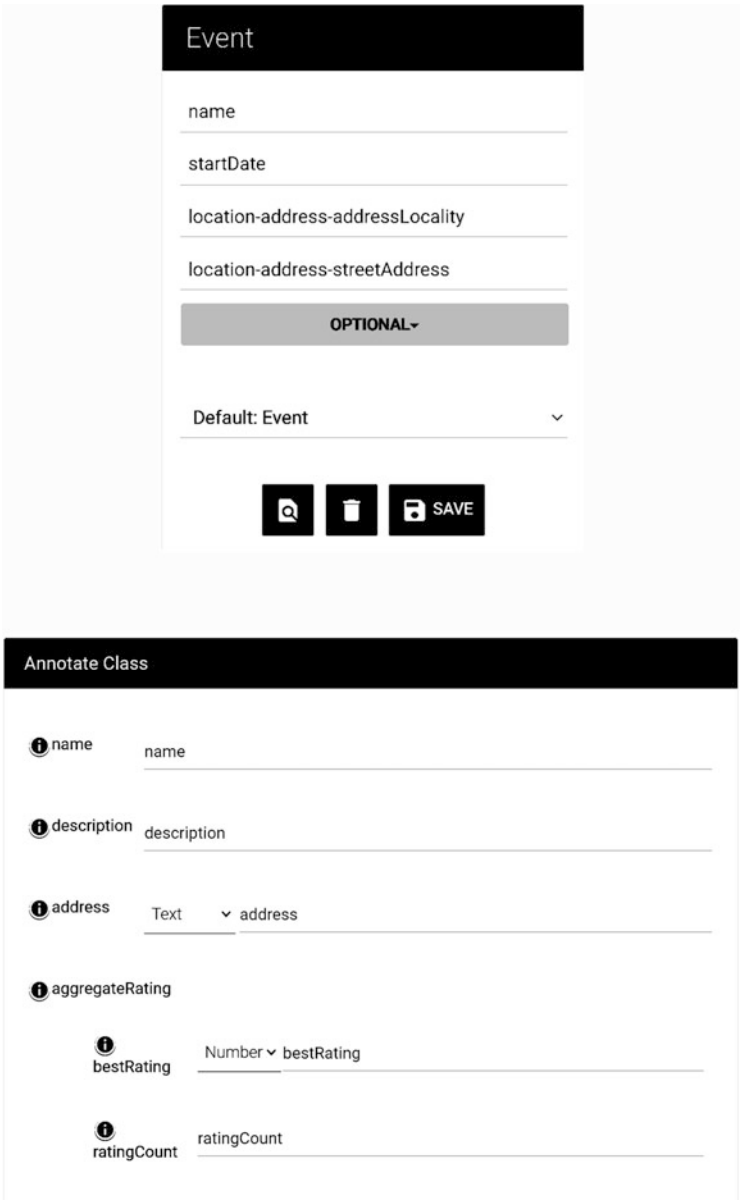


Fig. 23.1 Manual annotation editor

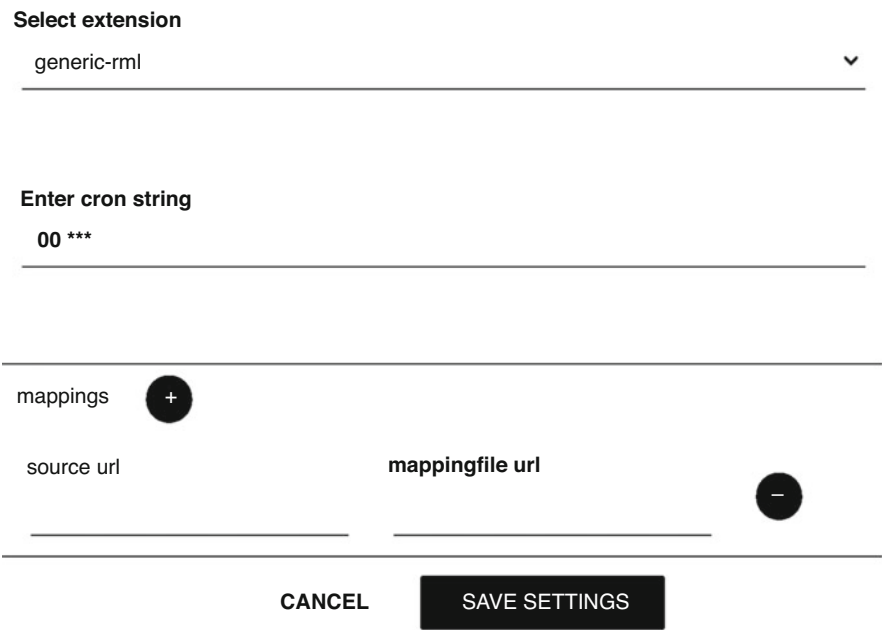


Fig. 23.2 RML mapping interface

tations are stored in a document store (MongoDB[8]) for efficient deployment as Web site annotations and a graph database (GraphDB[9]) (Bishop et al. 2011) for deployment as a full-fledged knowledge graph. The storage mechanism is shown in Fig. 23.3. In the remainder of this section, we will first cover the Web annotation scenario and then the knowledge graph scenario briefly. We refer the readers to Chap. 19 for a detailed explanation and comparison of the two approaches.

23.1.2.1 Hosting of Web Annotations

In this scenario, semantically annotated data are stored for usage on Web sites. This use case is particularly important for search engine optimization (SEO) activities and is the initial motivation of semantic annotations on the Web, particularly with schema.org. The created knowledge is stored as JSON for Linked Data (JSON-LD) documents in a document database. The main advantages of using such an approach are:

[8] https://www.mongodb.com/

[9] https://www.ontotext.com/products/graphdb/

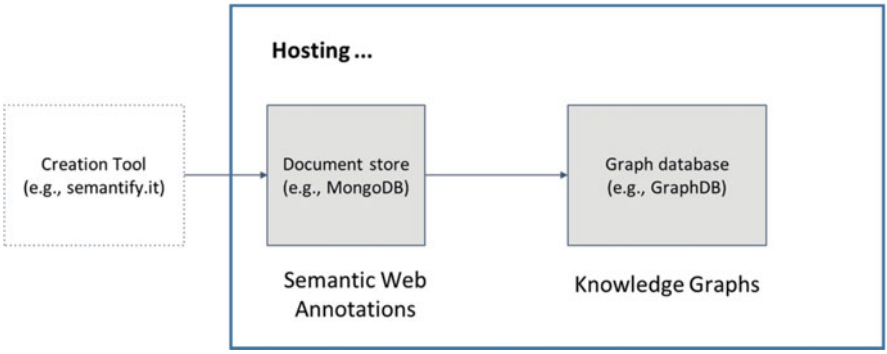


Fig. 23.3 Hosting architecture of Semantify.it

- Rapid storage and retrieval (through advanced JSON indexing)
- Cost-effectiveness thanks to many powerful open-source or free-to-use implementations
- Many libraries for Web development
- Comprehensive documentation and large user community

Hosting knowledge as a JSON-LD document is very efficient for the Web site annotation scenario as accessing an individual annotation to deploy it on a Web site can happen in $O(1)$ time.

23.1.2.2 Hosting of Knowledge Graphs

In this scenario, created knowledge is stored in an RDF triplestore. The scenario is particularly useful when there are advanced reasoning needs and intelligent applications are supposed to access integrated data. Triplestores are built on top of World Wide Web Consortium (W3C) recommendations, which means there are standard languages for data modeling, querying, and reasoning. Moreover, provenance information can be stored together with the data with the help of Named Graphs or RDF-Star.[10]

23.1.3 Knowledge Assessment

In Chap. 20, we introduced a knowledge assessment methodology that considers weights for dimensions and metrics and produces an aggregated quality score. Semantify.it offers the quality assessment tool (QAT).[11] QAT supports up to

[10] See Sect. 13.1 for details.

[11] https://qat.semantify.it

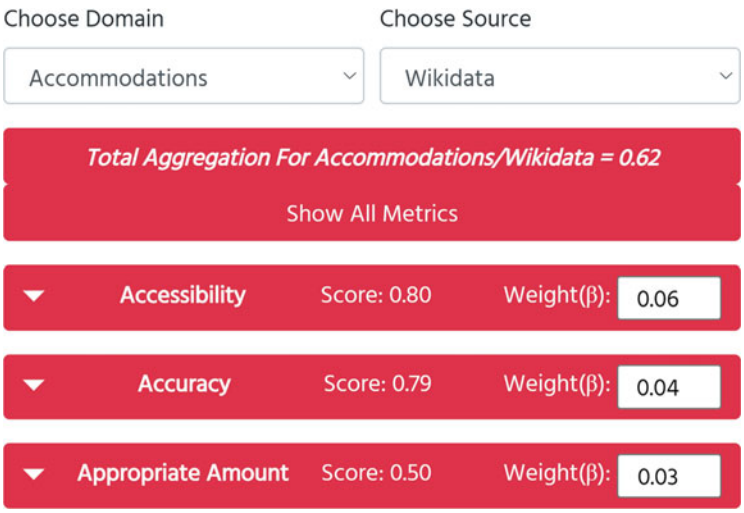


Fig. 23.4 An excerpt from the quality assessment tool interface

20 dimensions and 40 metrics in total. The interface allows weight definitions for each dimension and metric. The tool also provides an API for programmatical access for defining weights and making quality score calculations. Figure 23.4 shows the interface of QAT, where the weights for dimensions and metrics can be defined for various domains.

At the current stage, the knowledge graphs that can be accessed via QAT are statically defined in the interface (e.g., Wikidata, DBPedia, and LinkedGeoData) as some metric calculations require knowledge-graph-specific implementations. However, the work is ongoing for decoupling calculation function definitions for metrics from the system to allow a more flexible assessment process.

23.1.4 Knowledge Cleaning

In Chap. 21, we introduced error detection and correction approaches that provide support for the knowledge cleaning step in the knowledge graph life cycle. Semantify.it provides the evaluator module where semantic annotations on a Web site can be verified and validated. Remember that verification is done against a formal specification (e.g., an ontology, a set of integrity constraints, etc.), while validation is done against a domain of discourse. In the case of annotations of a Web site, the website is the domain.

Automated verification is rather straightforward to implement if the formal specifications are defined in advance. In the case of Semantify.it, verification is done against the schema.org vocabulary and domain specifications. Validation,

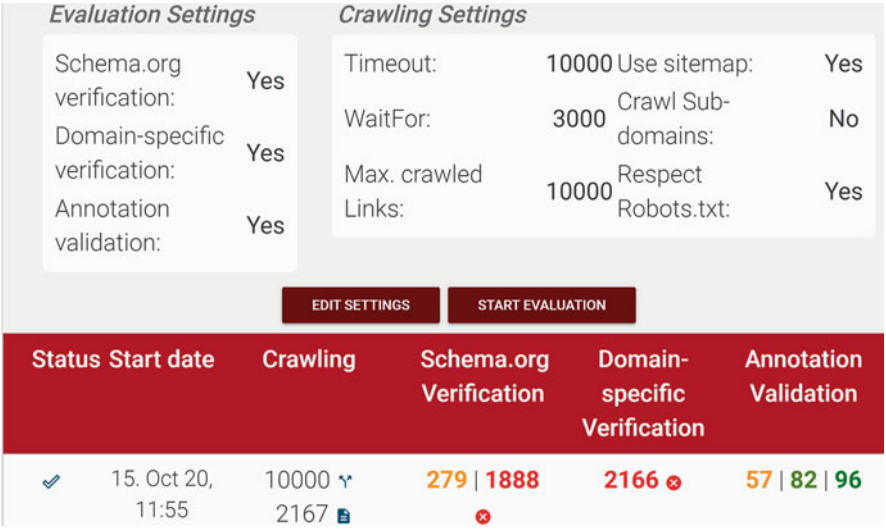


Fig. 23.5 Evaluator interface

however, is only feasible in such an automated setting for Web annotation scenarios as the semantic annotation on a Web page can be compared with the content of that Web page. The Web page here serves as the domain of discourse. Evaluating the domain of a knowledge graph is significantly harder.

Figure 23.5 shows the Evaluator interface. The Evaluator module takes a Web site Uniform Resource Identifier (URI) and some configuration parameters for crawling as input. It crawls the Web site according to those parameters and produces a report. The first row in the figure shows that the evaluation was successfully completed, which started on October 15, 2020, with a total of 10,000 pages crawled and 2167 annotations evaluated. Two hundred and seventy-nine of those annotations had a minor issue, like using HTTP URI for schema.org namespace instead of HTTPS. The remaining 1888 did not comply with the schema.org vocabulary (e.g., domain/range violations for property values). There was one annotation that did not have a matching domain specification on the Semantify.it platform. The others were verified against at least one domain specification, but they had verification errors. An annotation and a domain specification are matched based on their type. The annotation validation column shows up to three scores, worst score, average score, and best score, obtained by the annotations across all Web pages according to how well they represent the content they are annotating. One annotation scored 57%, which is the worst score; the average is 82%, and the annotation that best describes the Web page from which it is crawled scored 96%.

23.1.5 Knowledge Enrichment

Semantify.it does not currently provide an enrichment module out of the box. However, an external service can be used for knowledge enrichment hosted in the GraphDB installation in Semantify.it. A tool called Duplicate Detection as a Service (DDaaS) (Opdenplatz et al. 2022) is currently being developed, and a potential integration is planned for Semantify.it. The tool implements schema matching, indexing, prefiltering, normalization, comparison, and a decision process model presented in Chap. 22.

23.1.6 Knowledge Deployment

The Semantify.it platform makes the knowledge available in various ways. First, the annotations that are stored in the document store can be accessed through an HTTP API programmatically. Second, the collections of annotations are stored in Semantify.it GraphDB installation in a named graph and can be queried via a SPARQL endpoint. To facilitate these methods, Semantify.it provides the "pages" feature. A single-page Web site can be produced for each Semantify.it Web site (upon request). This single-page Web site also provides some statistics about the knowledge graph being deployed. Figure 23.6 shows some of the annotations created for tirol.at and the statistics about types that have the most instances. All annotations for tirol.at can also be downloaded from this page with one click.

The Web site also contains documentation for the HTTP API and provides an interface to query the SPARQL endpoint. Figure 23.7 shows the API endpoints, together with the opportunity to try them out based on the Swagger[12] API documentation and a SPARQL query editor.

23.2 Knowledge Access and Representation Layer

So far, we have discussed a knowledge graph life cycle for a single very large knowledge graph. However, the large and heterogeneous nature of knowledge graphs may cause issues when they are curated. Especially, the curation operations may not scale and become infeasible to run over billions of facts in a knowledge graph. Additionally, different applications and use cases may have different points of view in terms of the terminology they use and the constraints of their domain; therefore, conflicting schemas and constraints may need to be supported by the knowledge graph application.

[12] https://swagger.io/

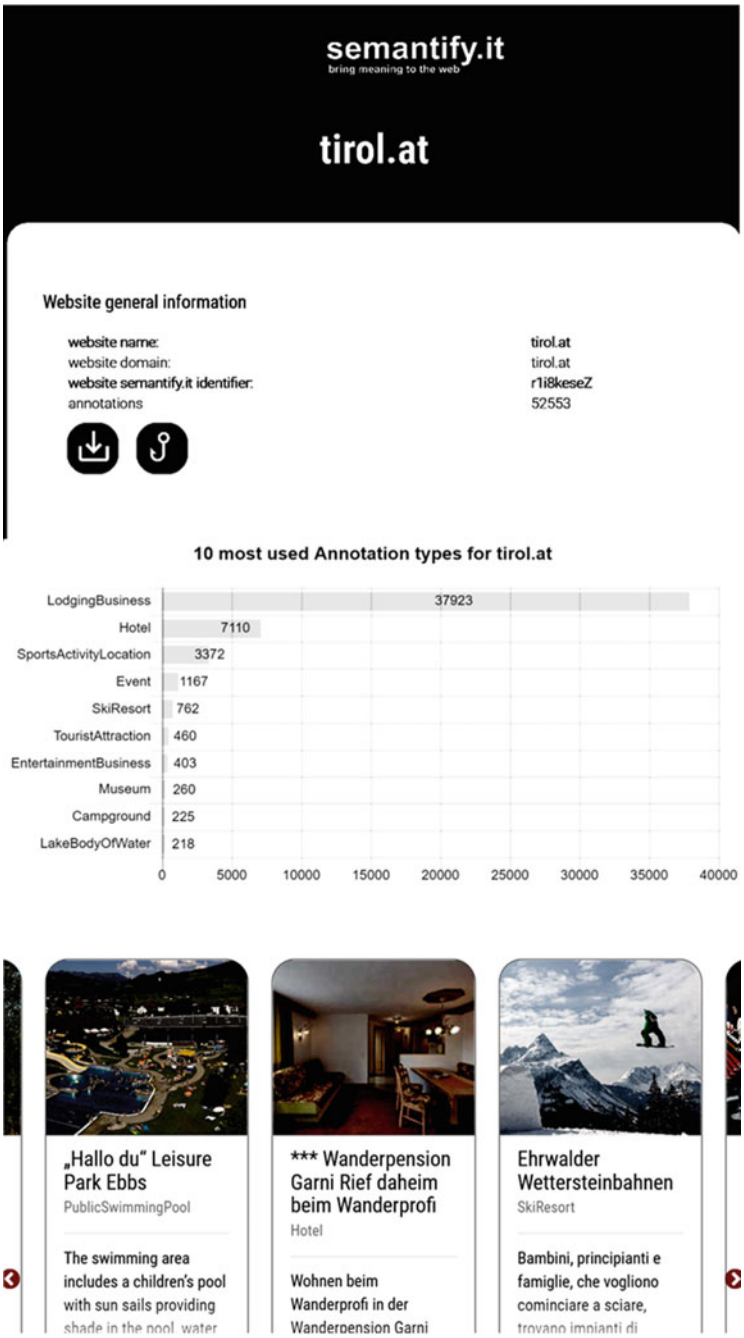


Fig. 23.6 Semantify pages as a deployment possibility

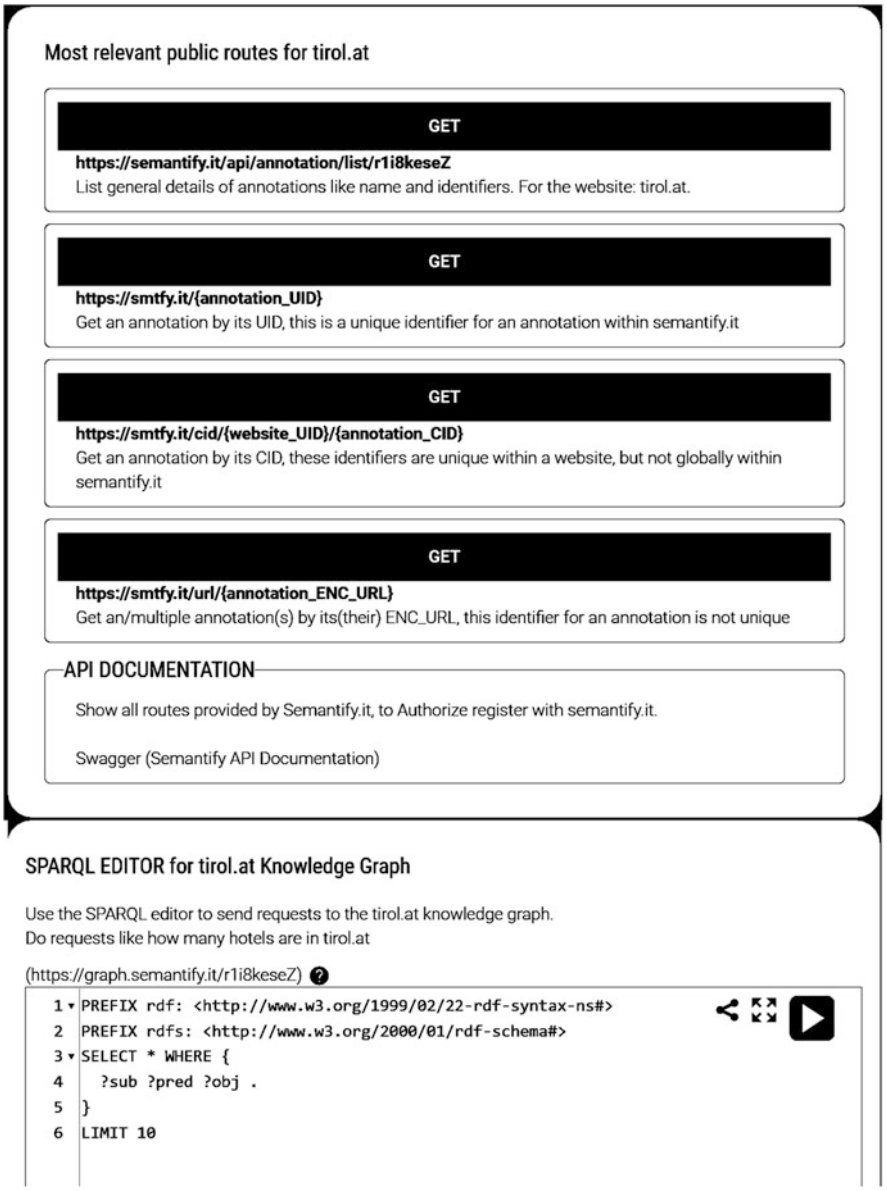


Fig. 23.7 API and SPARQL access to deployed knowledge

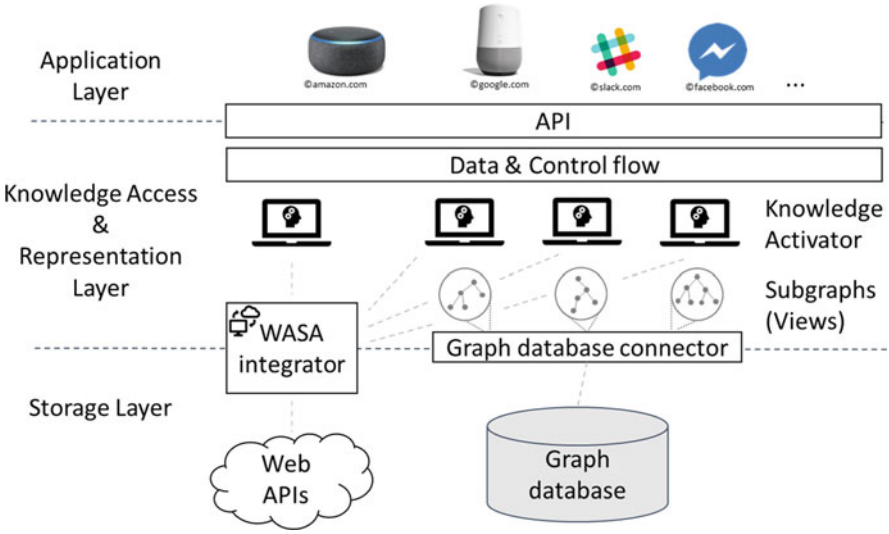


Fig. 23.8 The architecture of the Knowledge Access and Representation Layer

A middle layer called the *Knowledge Access and Representation Layer (KARL)* (Angele et al. 2022) has been developed to overcome these issues.[13] This layer provides different views over the knowledge graphs for two different reasons:

- The views only contain the relevant portion of the knowledge graphs for certain applications and use cases, reducing the size of the knowledge graph that needs to be curated, processed, and maintained significantly.
- And each view can have its own TBox to support different points of view on the knowledge graph for different applications and use cases.

In the remainder of this section, we introduce the overall architecture of such a layer, then introduce the core component of the layer, namely, *Knowledge Activators*, and provide an illustrative example.

23.2.1 Architecture

As shown in Fig. 23.8, the Knowledge Acquisition and Representation Layer (KARL) is situated on the top of a large, potentially erroneous and incomplete knowledge graph. We will explain the architecture bottom-up.

[13] The content of this section is largely based on the ongoing doctoral work of Kevin Angele. See Angele et al. (2022) for a summary.

A *graph database connector* component serves as a bridge between KARL and the underlying knowledge graph. It runs the queries sent by the Knowledge Activators to extract views.

A *view* is a (potentially modified) subgraph of the underlying knowledge graph and resides in a *Knowledge Activator*. A Knowledge Activator contains a view definition and stores that extracted view. It contains a specific TBox for that view, called Micro TBox, which contains terminology, constraints, and inference rules to support specific applications and use cases. It also contains engines for inference, duplicate detection, and error detection, which work on its view using the Micro TBox.

The data in a Knowledge Activator can be further enriched via *active and dynamic data*. The external data integrator component (e.g., WASA Integrator) uses semantic descriptions of Web services (e.g., with WASA (Şimşek et al. 2018)) (as described in Chap. 18) to bring dynamic data on demand or take actions like booking a hotel room or buying tickets for an event. The data brought in from external services are integrated and curated with the data in the relevant Knowledge Activators to answer application queries.

On top of the Knowledge Activators, there is the data and control flow. This component decides the flow of the data across Knowledge Activators and between the applications and the Knowledge Activators. Particularly, the data flow specifies the possible flow of data, and the control flow puts conditions on that flow. The control flow works based on the abstract state machine[14] (Gurevich 1995) principle, which means that while the system is in a certain state (e.g., certain property value assertions exist in the views, certain input values are provided by the user), the guarded transition rules decide the next state of the system.

Knowledge Activators reside in the core of KARL. Therefore, in the following two sections, we will cover the specifications and engines of the Knowledge Activators.

23.2.2 Knowledge Activators

Knowledge Activators contain a set of specifications and engines, as shown in Fig. 23.9. In the following, we will first briefly explain what every specification is for (Sect. 23.2.2.1) and how different engines make use of them (Sect. 23.2.2.2).

23.2.2.1 Specifications

Specifications are related to views in Knowledge Activators. Views are use-case or application-specific subgraphs extracted from the large underlying knowledge graph

[14] https://en.wikipedia.org/wiki/Abstract_state_machine

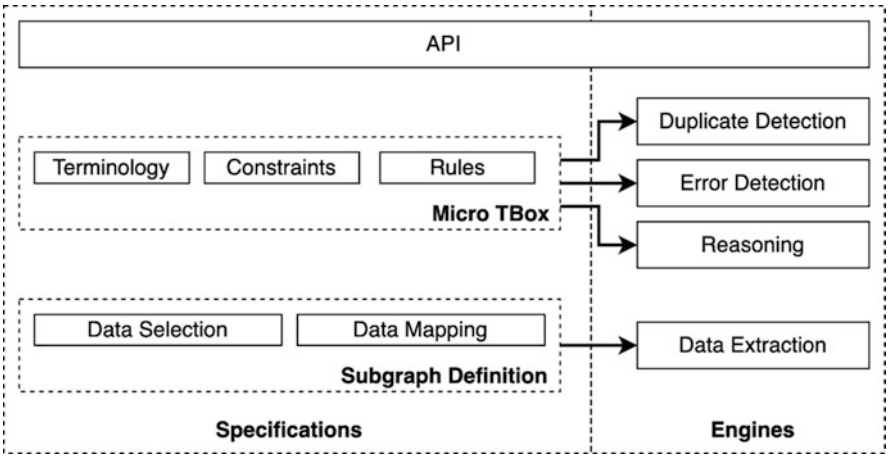


Fig. 23.9 The internal components of a knowledge activator

to reduce the amount of data to operate on. We examine the specifications in two categories:

- The specifications used to describe views (MicroTBox)
- The specifications used to define views (subgraph definitions)

A *MicroTBox* describes the Knowledge Activator from the point of view of a use case or application. It consists of three parts, namely:

- *Terminology* – types, properties, and the type hierarchy
- *Constraints* – certain requirements that instances/properties need to fulfill (e.g., the telephone number of a restaurant in Germany must start with "+49".)
- *Rules* – more specifically, inference rules that are used to infer new knowledge based on existing knowledge (e.g., if a restaurant serves only vegan dishes, then infer the type *VeganRestaurant*)

Although the view in a Knowledge Activator is a subgraph of the underlying knowledge graph, the MicroTBox is not necessarily a subset of the TBox of that knowledge graph as it may contain new terminology, constraints, and rules.

The *subgraph definition specifications* consist of two parts: data selection and data mapping. The data selection specification determines which subgraph of the underlying knowledge graph will be extracted. This specification is done via GraphQL[15] queries or directly via the WHERE clauses of SPARQL CONSTRUCT queries. The data mapping specification complements the data selection by mapping the TBox of the extracted view from the underlying knowledge graph to the Micro TBox defined by the Knowledge Activator. The data mappings can be specified as

[15] https://graphql.org/

part of the SPARQL CONSTRUCT query. We have seen how CONSTRUCT queries are used to map from one schema to another one in Sect. 13.2.

23.2.2.2 Engines

There are four engines that make use of the specifications we have covered so far: data extraction, reasoning, error detection, and duplication detection engines.

The *data extraction engine* takes a view (subgraph) definition and extracts a view from the underlying knowledge graph. This engine consists of a connector for the SPARQL endpoint and/or an RML mapper. The extracted data are mapped to the Micro TBox in the Knowledge Activator and then stored for the remaining three engines.

The *reasoning engine* is a rule-based reasoner to infer new knowledge based on the existing knowledge in a Knowledge Activator. It benefits from the terminology provided by the Micro TBox, such as the type hierarchy, as well as additional inference rules specific to a use case.

The *error detection engine* uses the terminology and the constraints in the Micro TBox to detect errors based on the requirements of a specific use case or application. This engine typically works as a SHACL verifier, as we have seen in Sect. 13.2.

The *duplicate detection engine* implements the duplication detection operation, as we discussed in Chap. 22. The difference this time is that the Micro TBox is used instead of the TBox of the underlying schema for configuration, and the operation is running over a much smaller knowledge graph. Moreover, any duplicates coming from external sources are handled by this engine (e.g., the same event instances may come from the knowledge graph but also from a Web service for events).

23.2.3 Illustration

In this section, we will provide a small example to show how Knowledge Activators are defined. The example is built around an application that serves the vegan community and that is only interested in vegan restaurants in Germany. The application is built on top of a large knowledge graph that contains tourism-related data such as local businesses, events, and points of interest (POIs)[16] (see also Şimşek et al. (2022)).

In the following, we will first give an example specification of a Knowledge Activator and then exemplify how a query runs through KARL based on the defined data and control flow.

[16] https://open-data-germany.org/open-data-germany/

23.2.3.1 A Knowledge Activator for Vegan Restaurants in Germany

In this section, we will specify a Knowledge Activator for vegan restaurants in Germany (*KA-VeganRestaurants*). A Knowledge Activator consists of two main parts, namely the specifications and engines. Figure 23.10 shows the process model involving the specification of a Knowledge Activator and its internal curation activities:

1. Define schema, constraints, and rules that describe the data in the view of a Knowledge Activator. The output of this step is the MicroTBox definition.
2. Define subgraphs and mappings to extract data from the large knowledge graph and map it to the terminology defined in the MicroTBox. The output of this step is the subgraph definition.
3. Extract subgraphs using the subgraph definitions from the previous step. The output of this step is a view.
4. Execute error detection and duplicate detection on the extracted views with the help of the terminology, constraints, and rules in the MicroTBox.

The section is structured based on these four steps. First, we will explain how the MicroTBox of *KA-VeganRestaurants* is defined. Then we will introduce the subgraph definition. Finally, we will briefly explain how the error detection and duplicate detection engines run on the view in this Knowledge Activator.

Define schema, constraints, and rules for MicroTBox. The MicroTBox consists of the terminology, an extended subset of the schema of the underlying knowledge graph, constraints that are specific to a use case, and inference rules to infer new knowledge based on the needs of a specific use case.

Terminology: Table 23.1 shows the terminology for the Knowledge Activator. For the terminology, a new *n:VeganRestaurant* type is introduced. This type is defined as a subtype of the *schema:Restaurant* type and inherits all its properties. The *n:VeganRestaurant* type also introduces a new property called *n:sealOfApproval*[17] with a *Text* range to indicate the seal of approval a restaurant has for serving certified vegan food (e.g., BeVeg[18]). Then a domain and use-case relevant subset of the n:VeganRestaurant type is selected.

Constraints: From the perspective of this use case, restaurants must be vegan; therefore, the following constraint as a SHACL shape can be defined: "A restaurant must have the value '*Vegan*' for the *schema:servesCuisine* property, and nothing else." Figure 23.11 shows the SHACL shape for this constraint. The shape applies to all schema:Restaurant instances. The possible values for the *schema:servesCuisine* property are restricted to a finite closed set that has only the "Vegan" value in it.

Rules: the Knowledge Activator in this example introduces one inference rule, which is shown in Fig. 23.12: "For all restaurants that have the value *BeVeg* in its *n:*

[17] *n* is the prefix of a newly created namespace.

[18] https://www.beveg.com/ – a certification program for service providers that do not use any animal-based product.

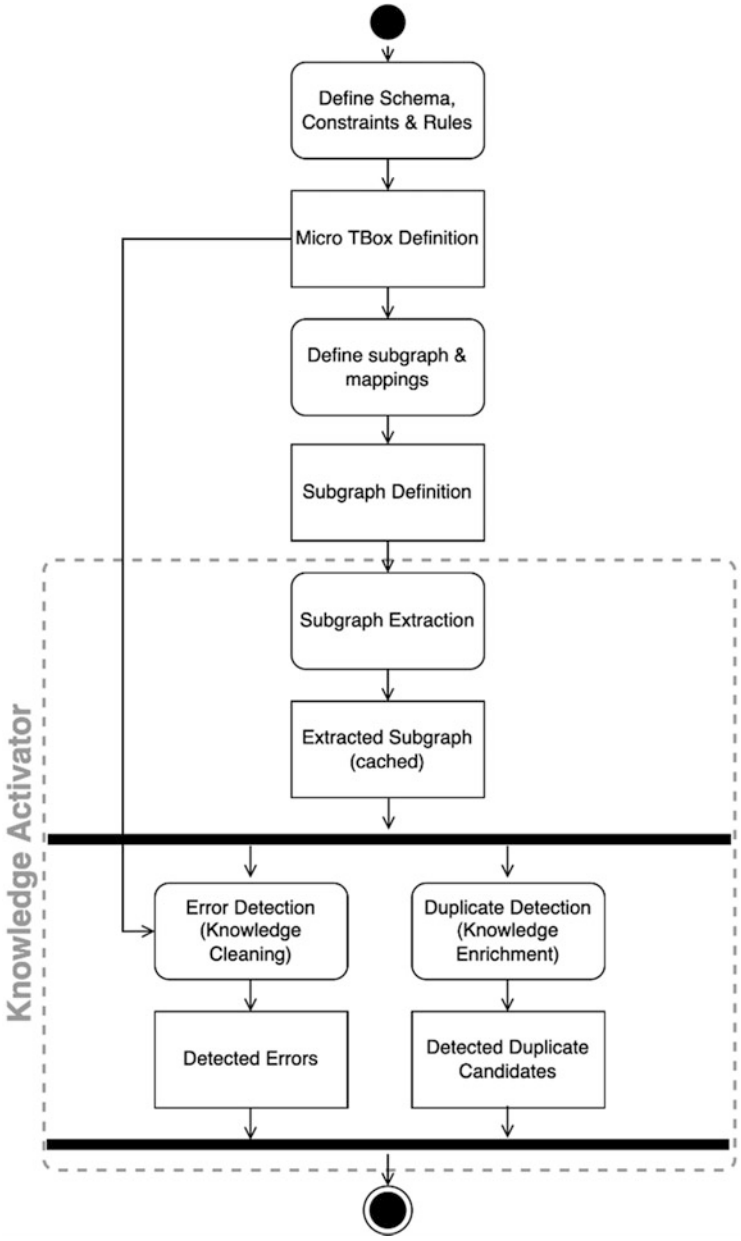


Fig. 23.10 The process model for specifying a knowledge activator and the curation process in a knowledge activator

sealOfApproval property, infer that they are vegan restaurants and they have the value '*Vegan*' for the *schema:servesCuisine* property." The rule is a deductive rule written in the F-logic syntax (see Sect. 13.3.2 for deductive rules and F-logic).

Table 23.1 Terminology for *KA-VeganRestaurant*

Thing > Place > LocalBusiness > FoodEstablishment > Restaurant > n:VeganRestaurant		
Property	Range	Description
schema:hasMenu	Menu or Text or URL	The actual menu
schema:location	Place or PostalAddress	The location of the restaurant
schema:name	Text	Name of the restaurant
schema: openingHourSpecification	OpeningHoursSpecification	The opening hours for the given restaurant
schema:servesCuisine	Text	The cuisine of the restaurant
n:sealOfApproval	Text	A seal of approval for the restaurant from a certification body

Define subgraphs and mappings. The second part of the Knowledge Activator specification is the subgraph definition. This definition consists of data selection and mapping.

For data selection and mapping, a CONSTRUCT query can be written to extract and map a subset of the underlying large knowledge graph. In this case, the WHERE clause of the query selects a subgraph, and the CONSTRUCT clause builds the subgraph and maps the selected subgraph's terminology, when necessary.

Figure 23.13 shows the CONSTRUCT query that extracts a subgraph from a knowledge graph and maps it to the terminology in the MicroTBox. This query retrieves all restaurant instances, with a subset of their property values for the properties defined in the MicroTBox (Table 23.1). The mapping is almost one to one, except for the parts that are marked in bold in the figure. Given that the application only serves in Germany, the query retrieves only the instances that have the *schema:addressCountry* value "Germany" and *schema:award* property value "BeVeg." The query also maps *schema:award* property to *n:sealOfApproval* property.

Extract a subgraph using the subgraph definition. The CONSTRUCT query in Fig. 23.13 is run on the large knowledge graph, and the resulting RDF graph is stored in the view of *KA-VeganRestaurants.*

Reasoning, error detection, and duplicate detection on the extracted views. The reasoner runs over the extracted instances to infer new values for the *schema: servesCuisine* property given the defined rule; the error detection engine verifies whether all instances fit the defined constraints (i.e., they are vegan restaurants) and produces errors when the constraints are violated. The instances that violate the constraints may be left out of the view. Additionally, the duplicate detection engine links duplicate restaurant instances if they occur. Note that all tools introduced in Chaps. 21 and 22 can be utilized, but given the smaller size of the views than the underlying knowledge graph, all these operations can run significantly faster and scale better.

```
@prefix ex: <http://example.org#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <https://schema.org/> .

ex:VeganRestaurantShape a sh:NodeShape ;
  sh:targetClass schema:Restaurant ;
  sh:property [
    sh:path schema:servesCuisine;
    sh:datatype xsd:string;
    sh:in ("Vegan")
  ] .
```

Fig. 23.11 A constraint that specifies every restaurant instance in this knowledge activator

$$X{:}\,n{:}\,VeganRestaurant[schema{:}\,servesCuisine \rightarrow \text{"Vegan"}] \leftarrow X{:}\,schema{:}\,Restaurant \wedge X[n{:}\,sealOfApproval \rightarrow \text{"}BeVeg\text{"}]$$

Fig. 23.12 An inference rule with the F-logic syntax to infer *VeganRestaurant* instances with the *servesCusine* property having the value "Vegan"

23.2.3.2 Running a Query Through KARL

After the Knowledge Activator specification process we described in the previous section, we end up with a Knowledge Activator that contains vegan restaurants in Germany. In this section, we will briefly demonstrate how a query coming from an application can run through KARL to obtain an answer.

The process model for answering a query with the help of KARL is shown in Fig. 23.14. Before the process starts, we assume the query is processed, and a structured representation of it is created (e.g., a natural language processing (NLP) engine extracted an annotation from the natural language query). First, the query is put into the "restaurant queries" input store. Then the control flow decides to which Knowledge Activator it is supposed to pass on the query result based on the guarded transition rules and the definitions of the data flow. At this stage, the Knowledge Activator runs the relevant part of the query on its view and returns the results. The results may come directly from the data stored in the view, or a Web service can be called (with the help of its semantic description). The results coming from the view and Web service are fused and then returned to the user (or to another Knowledge Activator, depending on the control flow).

To illustrate this process model, we will use the following scenario. Let us assume we have a KARL implementation with various Knowledge Activators, including

```
<Prefixes..>
CONSTRUCT
{
?r a schema:Restaurant;
  schema:hasMenu ?menu;
  schema:location [a schema:PostalAddress;
                   schema:addressCountry "Germany";
                   schema:addressLocality ?city];
  schema:name ?name;
  schema:openingHoursSpecification [a schema:OpeningHoursSpecification;
                                    schema:opens ?opening;
                                    schema:closes ?closing];
  schema:servesCuisine ?cuisine;
  n:sealOfApproval ?certificate .
}
WHERE
{
?r a schema:Restaurant;
  schema:hasMenu ?menu;
  schema:location [a schema:PostalAddress;
                   schema:addressCountry "Germany";
                   schema:addressLocality ?city];
  schema:name ?name;
  schema:openingHoursSpecification [a schema:OpeningHoursSpecification;
                                    schema:opens ?opening;
                                    schema:closes ?closing];
  OPTIONAL {schema:servesCuisine ?cuisine.}
  ?r schema:award ?certificate. FILTER(str(?certificate) = "BeVeg") .
}
```

Fig. 23.13 The CONSTRUCT query to extract a view from the knowledge graph for *KA-VeganRestaurant*

KA-VeganRestuarant, as specified in the previous section. The query "vegan restaurants in Germany" is sent by an application. The NLP engine processes this query and creates the structured representation in the JSON syntax, shown in Fig. 23.15.

The structured query is sent to the control and data flow in the next step. Figure 23.16 shows an excerpt from the data flow specification of the KARL implementation in this use case. Data flows are mainly used in software engineering to explicitly specify data flow between different processes. A Yourdon/DeMarco style notation (c.f. Yourdon (1989)) is used in Fig. 23.16. A data flow diagram (DFD) contains processes represented by ellipses, stores represented with two parallel lines that hold input and output of processes and arrows representing the

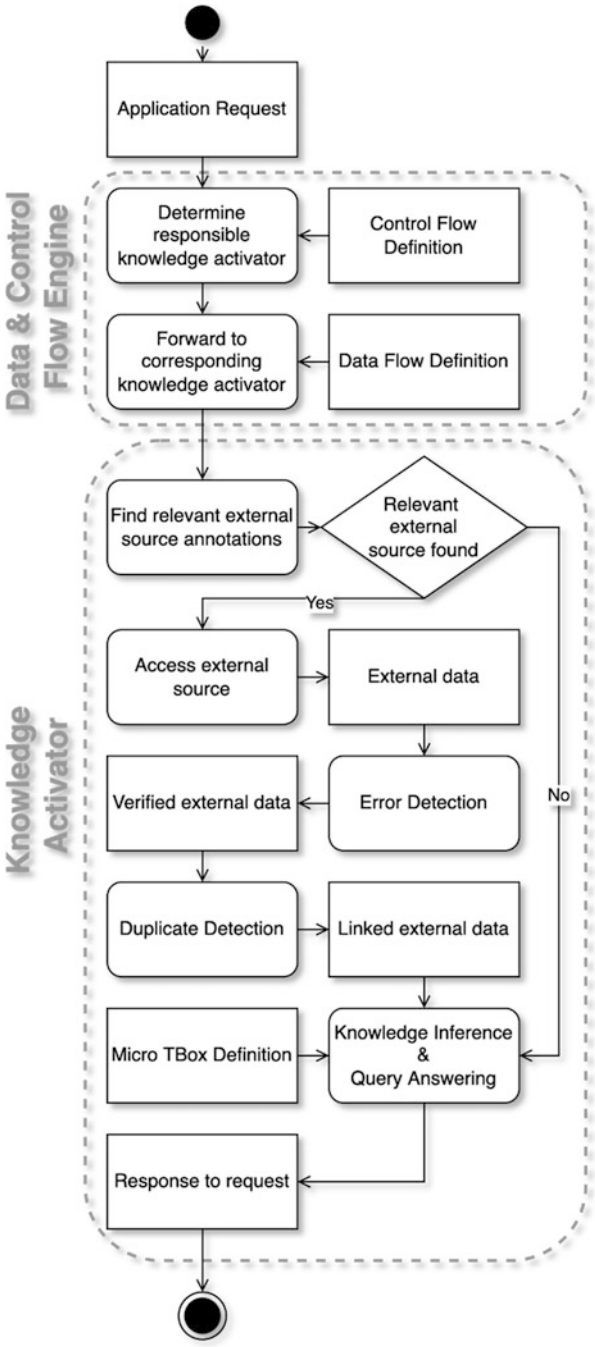


Fig. 23.14 The process model of KARL answering a query

```
{
        „@type“: „Restaurant“,
        „location“: {
                „addressCountry“: „Germany“
        },
        servesCuisine: „Vegan“
}
}
```

Fig. 23.15 The query object in the JSON syntax

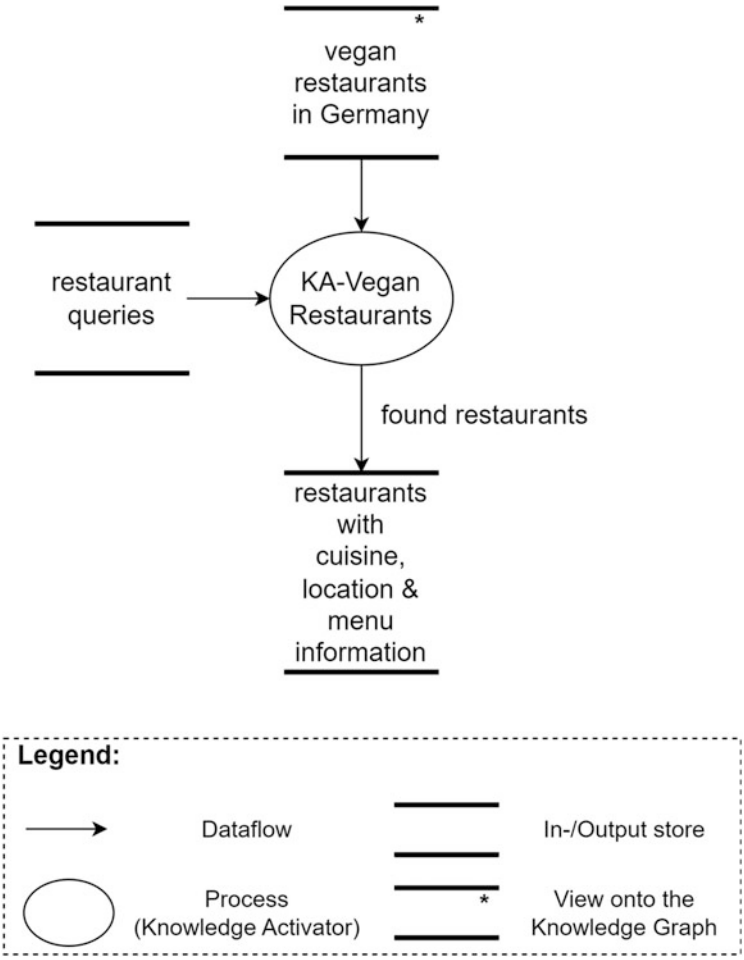


Fig. 23.16 A partial data flow specification

```
if (restaurantQueries.Restaurant(X).servesCuisine = Vegan AND
restaurantQueries.Restaurant(X).location.addressCountry = „Germany“)
then
        KA-VeganRestaurant(restaurantQueries.Restaurant(X),
veganRestuarantsInGermany)
```

Fig. 23.17 A guarded transition rule for the control flow

flow of the data between stores and processes. In this example, we have one process that corresponds to the *KA-VeganRestaurant* Knowledge Activator, two input stores, and one output store. One input store, "restaurant queries," holds the structured query provided by the user, and the other one, called *vegan restaurants in Germany*, is the view stored in *KA-VeganRestaurant*. The Knowledge Activator takes the query and stores the vegan restaurant instances with name, full address, cuisine, and menu information into the output store.

The data flow specifies the possible flows between stores and processes but does not specify under which condition the data should flow. For example, the restaurant query store may have an outgoing flow to different Knowledge Activators about restaurants. In such scenarios, a control flow is needed.

Assume that we have a control flow specification with guarded transition rules (an abstract state machine implementation), as shown in Fig. 23.17. The rule shown in the figure checks whether the state of the KARL determined by specific data stores (i.e., *restaurantQueries*) has a Restaurant instance with "Vegan" as the *servesCuisine* value and "Germany" as the *addressCountry* value. If this is the case, then the *KA-VeganRestaurant* Knowledge Activator is invoked (with the input stores *veganRestaurantsInGermany* and *restaurantQueries* as parameters).

After the query is processed via the *KA-VeganRestaurant* Knowledge Activator, the results retrieved from its view are stored in its output store and can be returned to the application. In more complex cases, the stored output can be used as an input for another Knowledge Activator if the current state (stored in the stores) triggers another guarded transition rule.

23.3 Summary

The final task in the knowledge graph life cycle, *knowledge deployment*, deals with making curated knowledge available for applications. In this chapter, we presented some tooling that helps deployment covering the entire knowledge life cycle to a large extent. The platform we focused on, *Semantify.it*, offers different modules to tackle different knowledge graph life cycle tasks. It targets two different kinds of

deployment, namely, Semantic Web site annotations and full-fledged knowledge graphs. We explained how knowledge could be deployed in both ways via Semantify.it.[19]

Naturally, deploying a knowledge graph as a large single entity comes with its own challenges. The curation operations may take a very long time, even may not scale at all, and may generate astronomic costs for millions and billions of statements due to the large size of a knowledge graph. Moreover, there can be different points of view on different applications and use cases in terms of terminology, constraints, and inference rules, which may not be possible to accommodate in a single large knowledge graph. To overcome these issues, we proposed a middle layer called the *Knowledge Access and Representation Layer (KARL)*, which enables the definition of use-case-specific views and terminologies, constraints, and inference rules. These reside in components called *Knowledge Activators.* Each Knowledge Activator provides a use-case-specific view that is significantly smaller than the underlying knowledge graph and a Micro TBox describing the view. A MicroTBox may contain terminology, constraints and rules that are use case specific and even conflicting across different use cases.

References

Angele K, Simsek U, Fensel D (2022) Towards a knowledge access & representation layer. In: SEMANTiCS (Posters & Demos), CEUR-WS.org, CEUR Workshop Proceedings, vol 3235

Bishop B, Kiryakov A, Ognyanoff D, Peikov I, Tashev Z, Velkov R (2011) OWLIM: a family of scalable semantic repositories. Semant Web 2(1):33–42

Cimmino A, Garcia-Castro R (2023) Semantic Web (accepted in January 2023, in press). https://www.semantic-web-journal.net/content/helio-framework-implementing-life-cycle-knowledge-graphs-0

Gurevich Y (1995) Evolving Algebras 1993: Lipari guide, specification and validation methods. Oxford University Press, pp 9–36. https://www.microsoft.com/en-us/research/publication/103-evolving-algebras-1993-lipari-guide

Kärle E, Simsek U, Fensel D (2017) semantify.it, a platform for creation, publication and distribution of semantic annotations. In: Proceedings of SEMAPRO 2017: the eleventh international conference on advances in semantic processing, Barcelona, November 25–29, pp 22–30

Opdenplatz J, Şimşek U, Fensel D (2022) Duplicate detection as a service. arXiv preprint arXiv:220709672

Şimşek U, Kärle E, Fensel D (2018) Machine readable web APIs with schema.org action annotations. Procedia Comput Sci 137:255–261

[19] Note that Semantify.it was a product of an industrial research cooperation and is currently being maintained by Onlim GmbH and offered as free to use. By the time you are reading this book, there can be some new features or some deprecated ones.

Şimşek U, Kärle E, Fensel D (2019) RocketRML - a NodeJS implementation of a use-case specific RML mapper. In: Proceedings of 1st knowledge graph building workshop co-located with 16th Extended Semantic Web Conference (ESWC), Portoroz, Slovenia, June 3, 2019. CEUR vol. 2489

Şimşek U, Kärle E, Angele K, Huaman E, Opdenplatz J, Sommer D, Umbrich J, Fensel D (2022) A knowledge graph perspective on knowledge engineering. SN Comput Sci 4(1):16

Yourdon E (1989) Modern structured analysis. Prentice-Hall

Chapter 24
Summary



Building a knowledge graph is not a one-off activity but a life cycle consisting of various processes. Figure 24.1 shows the knowledge graph life cycle tasks we covered in Part III. The life cycle mainly consists of *creation*, *hosting*, *curation*, and *deployment* tasks.

Knowledge creation is the first step, where heterogeneous, static, dynamic, active, unstructured, semi-structured, and structured data sources are integrated into a knowledge graph. Ontologies are at the core of the creation task. They describe the meaning of the data in the knowledge graph. An ontology mainly comprises the TBox of a knowledge graph. The techniques used in the creation process vary depending on the nature of the data source. Unstructured data sources such as text and images must be processed using information extraction techniques (e.g., via natural language processing (NLP) and image processing) to create semantically annotated data with the terminology provided by an ontology. Semi-structured and structured data sources already provide some sort of metadata; therefore, the knowledge creation process is mainly involved with the mapping of this metadata to an ontology and automatically generating a large number of facts (ABox).

The *knowledge hosting* step stores the created knowledge in a data container, usually a database. There are various ways to implement a storage facility for knowledge graphs, such as the relational, document, or graph model. The relational model is well established, and there is significant know-how and tool support for it. However, its nature is not always suitable for storing a knowledge graph efficiently and effectively since the schema it provides is very strict. For example, adding a new type or property to a knowledge graph may require that the entire schema of the relational database be recreated. Document stores are more flexible; however, the document model struggles to represent the connections between different nodes in a knowledge graph as it is not always clear how much information about an instance a document should contain. Graph databases provide many more native storage options for knowledge graphs. Property graph and Resource Description Framework (RDF) triplestores are the most popular graph database paradigms, with various advantages. RDF triplestores are built on strong standardization.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_24

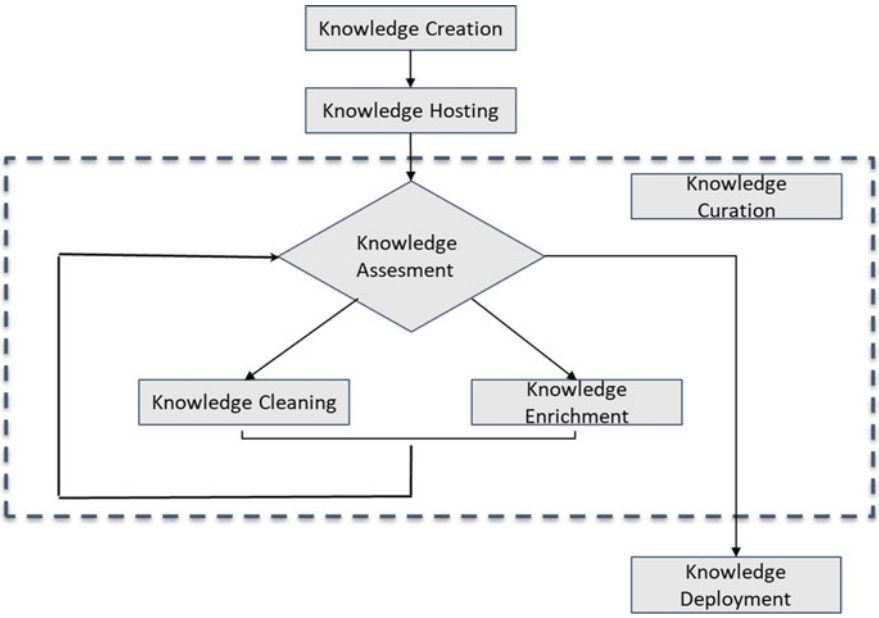


Fig. 24.1 Knowledge graph life cycle process model revisited

Knowledge graphs are large semantic nets built from heterogeneous sources. Their flexibility comes with the price of quality assurance as it is inevitable to have quality issues with such vast amounts of data. The *knowledge curation* task deals with the assessment and improvement of quality issues, particularly for correctness and completeness.

- The first task of knowledge curation is *knowledge assessment*. This task calculates and aggregates quality scores based on various metrics that belong to different dimensions. Each dimension and metric is weighted, which allows us to determine the most important ones for a domain and task. Depending on the assessment process results, some dimensions may need improvement to enhance the knowledge graph's overall quality.
- *Knowledge cleaning* is the task that aims to improve the correctness of a knowledge graph. It consists of error detection and error correction. The error detection task deals with identifying errors, either syntactically or semantically. Syntactic errors mainly occur from typos in resource identifiers and literal values, but they could also occur due to the broken serialization of RDF graphs. Semantic errors can be considered under two categories: errors due to the violation of formal specifications and errors due to the violation of facts in the domain of discourse. The former is identified through a verification process. The verification of a knowledge graph is done against a formal specification like integrity constraints; therefore, it is clear how to automate it. For example, identifying domain or range errors for a property is a verification activity. Errors in modeling the domain of

discourse must be identified through a validation process. The validation of a knowledge graph must be done against the domain it describes, and it is not always straightforward to automate. For example, a restaurant's telephone number may only be validated by calling the number and ensuring that the number belongs to that restaurant. The identified errors are then corrected manually or semi-automatically, depending on the nature of the error. Some errors can be fixed by using some domain knowledge (as we have seen in our geocoordinate example in Chap. 21) or via a statistical analysis of the existing knowledge graph, but some errors need to be fixed manually.

- *Knowledge enrichment* is the task that aims to improve the completeness of a knowledge graph. The first challenge to address while improving the completeness of a knowledge graph is to define what "complete" means. This is not always straightforward and usually depends on the context. Therefore, completeness is typically defined according to a domain, a task, or an application. For example, completeness can be defined in terms of the number of properties of a type that have value assertions or the number of queries that an application was able to answer powered by a knowledge graph. The enrichment process consists of three steps: selecting and mapping of an external data source, detecting duplicate instances, and the fusion of duplicate instances. Once a data source is identified and included in a knowledge graph, the duplicate instances are identified based on various similarity calculations between properties. Once duplicate entities are linked, the property values must be fused. This may lead to conflicting property values and a violation of formal specifications (e.g., functional properties having two distinct values). In this case, knowledge cleaning could be triggered to identify and resolve these errors.

The final step in the knowledge graph life cycle is the *deployment task*. The deployment task deals with making a knowledge graph available for various applications. This task involves tooling that provides Application Programming Interface (API) access and query interfaces to enable different access modalities. There are two major challenges while deploying a knowledge graph: the first is related to its large size. The bigger a knowledge graph gets, the more cumbersome the curation processes and query answering become. The second challenge is related to the different points of view of different applications and use cases. For example, different applications may use different terminologies and possibly conflicting constraints. These two challenges can be tackled with a Knowledge Access and Representation Layer that creates *views* on top of a large, possibly erroneous, and incomplete knowledge graph. These views are significantly smaller than the underlying knowledge graph, and they are described with their own terminology, constraints, and rules, called a MicroTBox.

In the next part, we will cover the applications that make use of curated and deployed knowledge graphs and some recent trends around knowledge graph technology.

Part IV
Applications

Chapter 25
Applications



So far, we have covered many theoretical topics around knowledge graphs and the technical details of how to model and implement them. However, building a knowledge graph is a challenging and typically costly endeavor. Therefore, knowledge graphs are constructed not just for the sake of making them, but they are also meant to power various applications.

Covering the entire landscape of applications for knowledge graphs would require hundreds of pages, and even then, we would not be achieving complete coverage. There are several surveys that examine applications powered by knowledge graphs from different perspectives. One example is the survey in Zou (2020), which classifies the applications into categories like question answering, recommender systems, and information retrieval applications in domains like health, news, finance, and cybersecurity.

In this chapter, we briefly introduce four major application areas of knowledge graphs that are increasingly ubiquitous and have a significant impact on everyday life.[1] We will first cover the application in the field of search engines, which turn them into query-answering engines, and then show how virtual assistants benefit from these knowledge graphs. Next, we introduce enterprise knowledge graphs and finally mention the recent developments in the adoption of knowledge graphs in cyber-physical systems.

25.1 Migration from Search to Query Answering

As we already briefly covered in Part I, the initial principle of search engines was the retrieval of Web documents based on statistical information retrieval methods, which required users to go through the search results to find the answers they had been

[1] With the danger of being utterly subjective.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

U. Serles, D. Fensel, *An Introduction to Knowledge Graphs*,
https://doi.org/10.1007/978-3-031-45256-7_25

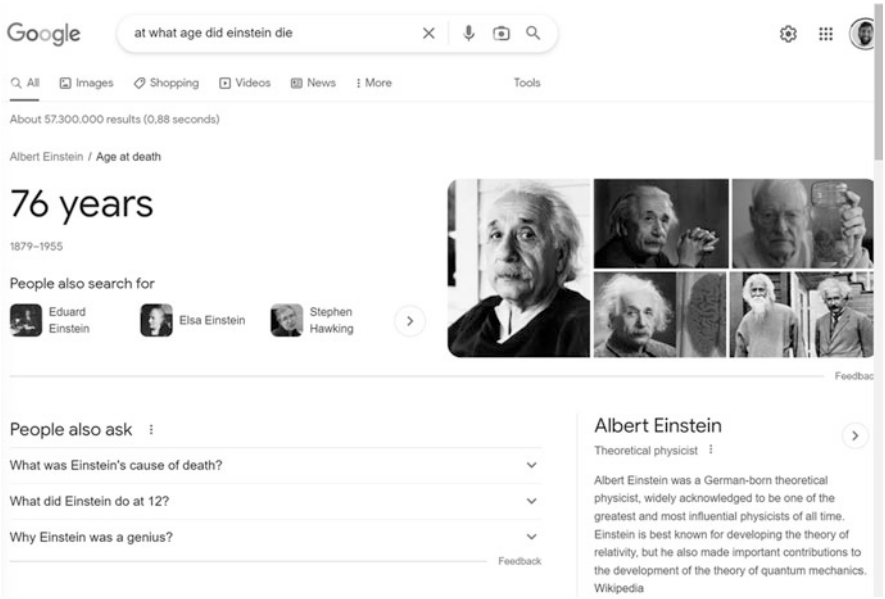


Fig. 25.1 Google as a query answering engine

looking for. This paradigm has been shifting since 2012 as search engines started to strive to keep users on their platforms instead of redirecting them to other Web sites. This was only possible if the search engines provided the answers for users' queries directly on the search result page and not merely list a bunch of Web pages that "may" have the answer.

Knowledge graph technology is the driver of the migration of search engines toward a query answering engines. Perhaps the most prominent application in this category is Google. Thanks to its Google Knowledge Graph, which is created from heterogeneous sources,[2] including billions of semantically annotated Web pages with schema.org, user queries can be answered without users even seeing a single Web page outside of Google. The example in Fig. 25.1 shows Google answering the question, "At what age did Einstein die?"

Meanwhile, other major search engines also go in a similar direction. Figure 25.2 shows Microsoft Bing answering the same question.[3] Microsoft Bing benefits from the Microsoft Knowledge Graph, which contains billions of statements,[4] like

[2] https://support.google.com/knowledgepanel/answer/9787176?hl=en

[3] https://techcommunity.microsoft.com/t5/microsoft-bing/microsoft-bing-is-becoming-more-visual/m-p/2200139

[4] https://blogs.bing.com/search-quality-insights/2017-07/bring-rich-knowledge-of-people-places-things-and-local-businesses-to-your-apps

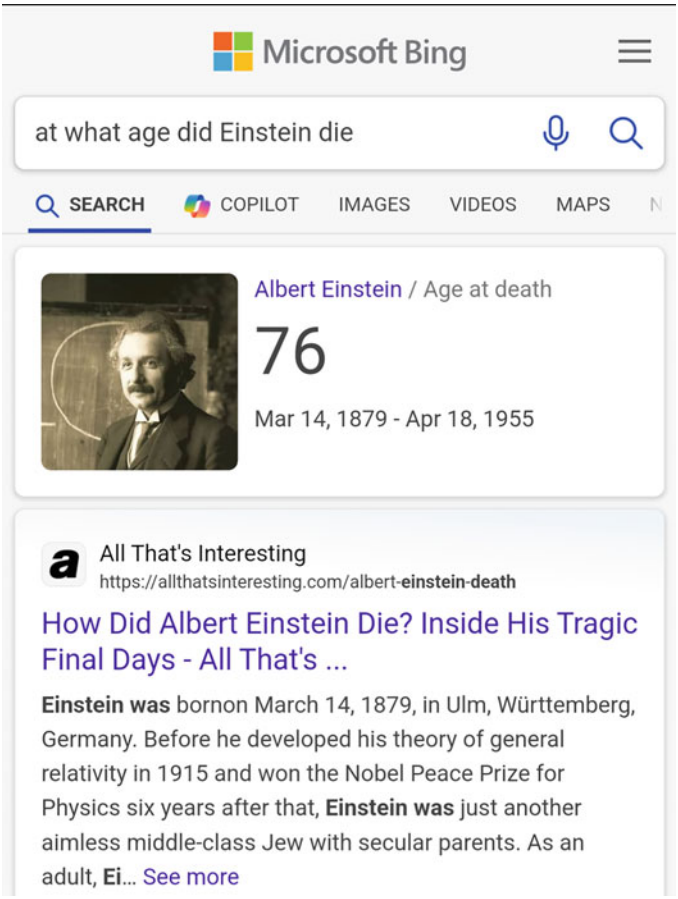


Fig. 25.2 Bing Knowledge Card for Albert Einstein

Google. Yahoo! also develops a knowledge graph to power various applications, such as Yahoo Finance, News, and Search.[5]

25.2 Virtual Assistants

Major tech companies have shown rapid development in intelligent virtual assistants that take spoken or written natural language commands and carry certain tasks on behalf of their users. Integrated with devices like smartphones, virtual assistants are

[5] https://developer.yahoo.com/blogs/616566076523839488/

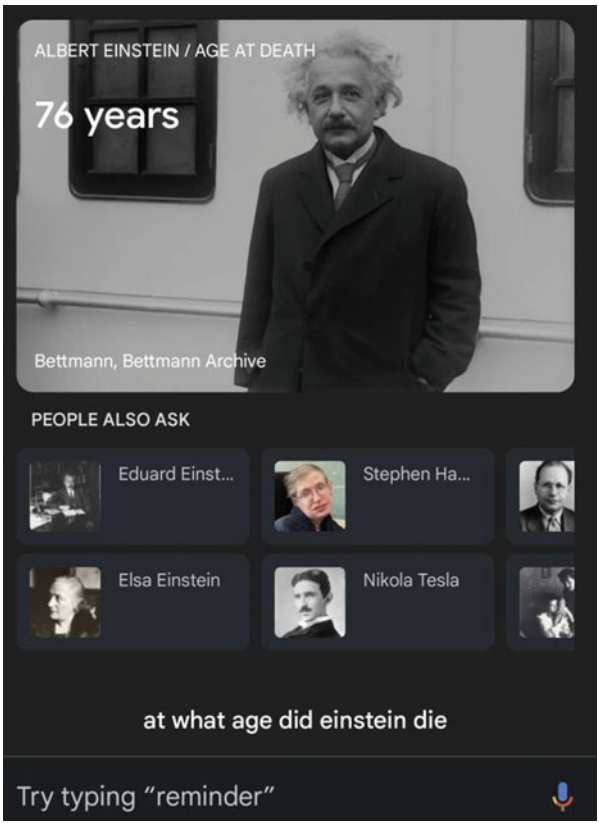


Fig. 25.3 Google Assistant answers questions with the help of Google Knowledge Graph

perhaps one of the most ubiquitous applications of artificial intelligence (AI). However, the success of a virtual assistant typically depends on two factors:

- How well does the assistant understand the natural language commands and generate natural language responses? This factor has been addressed intensively, and it is almost safe to say that the technology has become mainstream, thanks to developments in machine learning (ML) and increasing computational power.
- How much knowledge does the assistant need to have to perform its tasks properly? Here, knowledge graph technology is crucial.

Figure 25.3 shows an answer to the same question as in Sect. 25.1 from Google Assistant. The Google Knowledge Graph has powered Google Assistant since its inception and serves as a vast knowledge source.

Following Google's steps, many other virtual assistant providers started to implement knowledge graphs as a knowledge source for their applications. Apple is actively developing a knowledge graph under the Apple Knowledge Platform



Fig. 25.4 Question answering over semantic data on the Web

project to power Siri (e.g., Ilyas et al. 2022). Similarly, Amazon provides the possibility to power Alexa Skills with the Amazon Knowledge Graph.[6,7]

There are also smaller companies that build their conversational products completely on knowledge graphs. Onlim, an Austrian start-up and a spin-off company of the University of Innsbruck, uses knowledge graphs to automate online e-commerce and marketing via conversational agents in domains like education, energy, and tourism. Alongside using knowledge graphs as a knowledge source for the dialog systems, another major research and development goal is to automate conversational agent development as much as possible with the help of knowledge graphs. Knowledge graphs are used as training data for natural language understanding, as well as for the semi-automated generation of dialog systems via the extraction of capabilities (intents) for a dialog system from the semantic description of data and services[8] (Şimşek and Fensel 2018). A similar approach that uses ontologies to generate Alexa Skills can also be seen in Pellegrino et al. (2021).

Question answering over semantic data has been one of the major research interests since the beginning of the Semantic Web (Fig. 25.4). Such systems aim to use natural language questions to retrieve information from Resource Description Framework (RDF) data. In principle, these systems follow the following pipeline:

- Running typical natural language processing (NLP) tasks for a syntactic analysis of the question
- Entity mapping and disambiguation over the knowledge graph, and
- Query construction (e.g., SPARQL), query execution, and answer provision

Text and voice-based interaction with users has become mainstream; however, use cases are still basic (see Fig. 25.5). Without knowledge, there is no understanding of users' needs and goals. Only knowledge graphs that contain world knowledge can improve this and provide meaningful dialogs (Fig. 25.6).

A striking example of how important knowledge is for virtual assistants is the popular ChatGPT application from OpenAI.[9] ChatGPT is an application of the large language model called GPT (Generative Pretrained Transformers) (Radford et al. 2018). A language model[10] like GPT assigns probabilities to word sequences, which gives it the capability of generating impressively plausible sounding texts, given that

[6] https://developer.amazon.com/en-US/blogs/alexa/alexa-skills-kit/2021/02/alexa-entities-beta

[7] https://www.aboutamazon.com/news/devices/how-alexa-keeps-getting-smarter

[8] See also http://dialsws.xyz for examples

[9] https://openai.com/

[10] https://en.wikipedia.org/wiki/Language_model

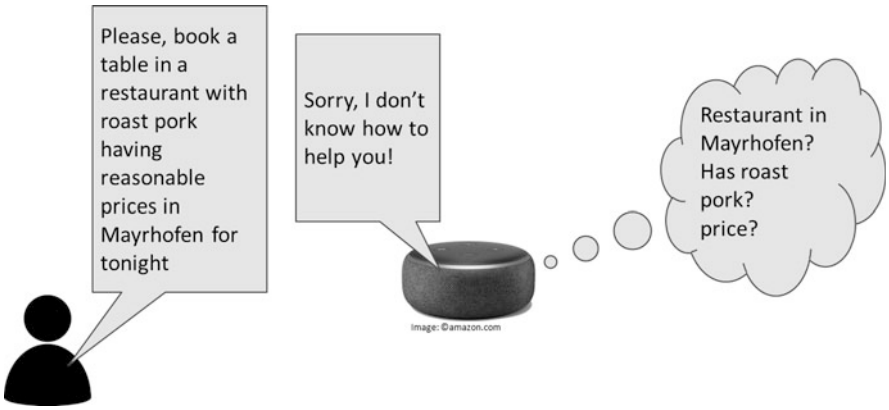


Fig. 25.5 No knowledge no meaningful dialog

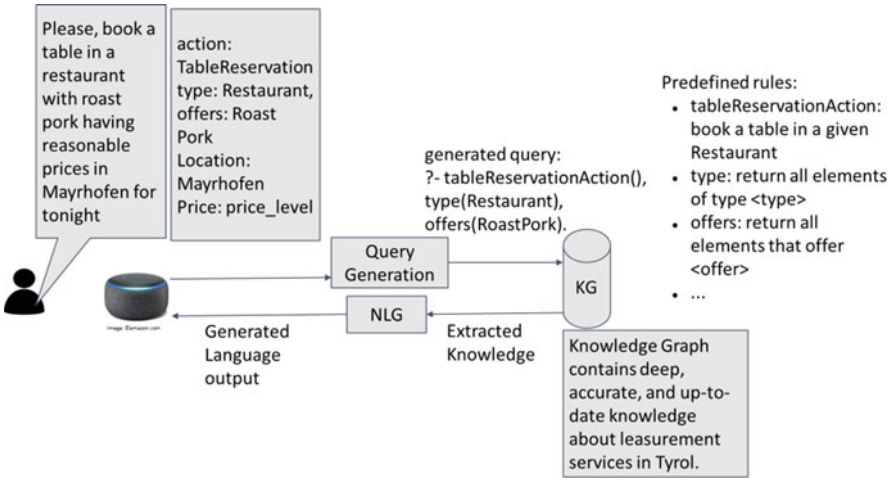


Fig. 25.6 A meaningful dialog requires world knowledge

a large training corpus is provided. The latest version of OpenAI's GPT model was trained on Common Crawl,[11] Webtext2 (Radford et al. 2019),[12] and Wikipedia, as well as some book datasets, which in total contain around 500 billion tokens.[13] Although ChatGPT brings significant improvements in addressing the natural language generation and understanding task, its lack of knowledge prevents it from being a viable virtual assistant option. For example, a query like

[11] https://commoncrawl.org/

[12] https://gregoreite.com/webtext2-webtext-openwebtext-inside-the-ai-datasets/

[13] https://en.wikipedia.org/wiki/GPT-3

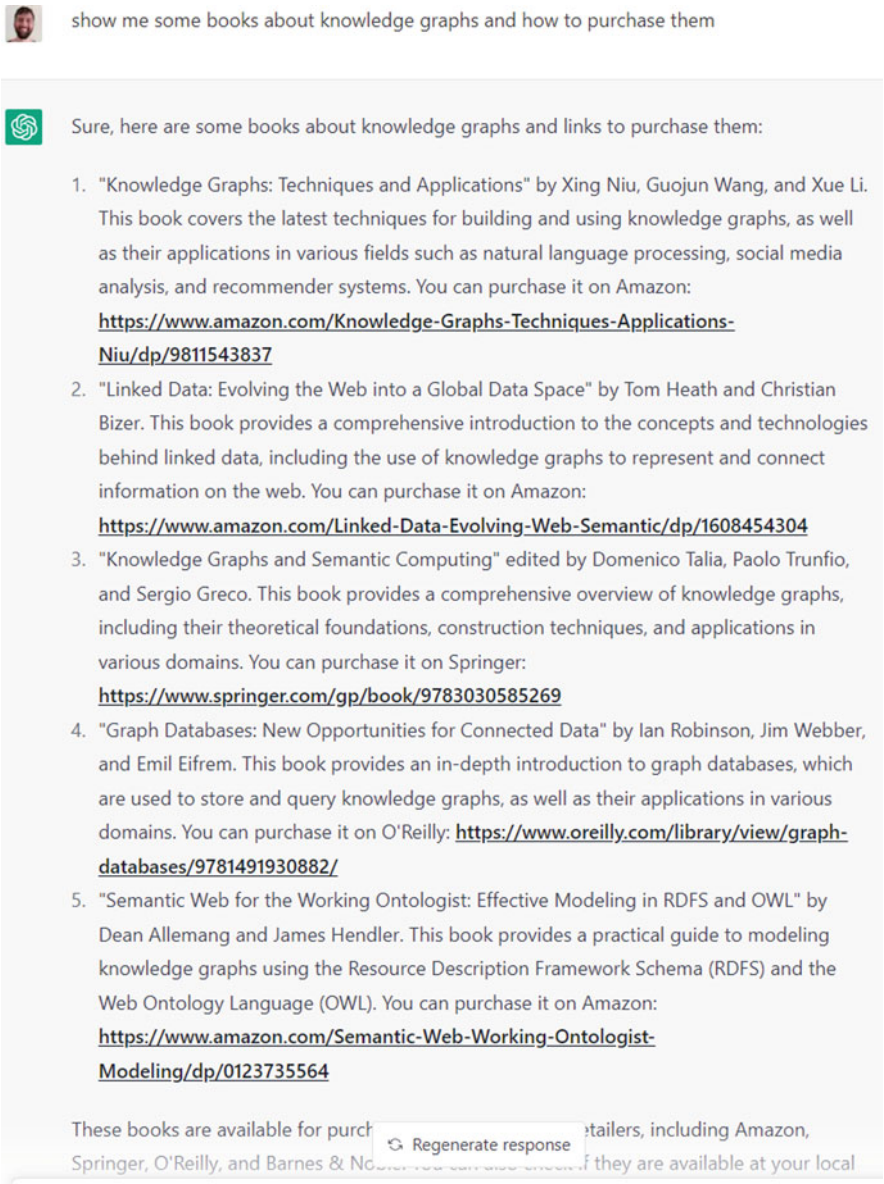


Fig. 25.7 A query to ChatGPT and its answer (Retrieved on 20.02.2023)

Show me some books about knowledge graphs and how to purchase them.

is answered with a set of book titles, their authors, and links to purchase them (Fig. 25.7). We receive five recommendations, and just one or two of them are actual

books with the correct authors/editors. One answer that ChatGPT gives is the following (answer number 3 in Fig. 25.7):

> "Knowledge Graphs and Semantic Computing" edited by Domenico Talia, Paolo Trunfio, and Sergio Greco. . .. You can purchase this book on Springer's Web site at https://www.springer.com/us/book/9783030651375.

The answer looks entirely plausible until you validate it: there is a conference proceeding that contains "Knowledge Graphs and Semantic Computing"[14] in its title, with a completely different set of editors. Moreover, the link to purchase the book does not exist. A similar query can be written for many things that someone could expect from a virtual assistant, e.g., events and products, and the results may be the same. This is because ChatGPT has no explicit knowledge about these entities but only guesses the most probable sequences of words as an answer to the query. Related to our example, books are typically well annotated[15] across the Web; therefore, a virtual assistant enriched with the knowledge collected from semantic annotations about books should have better accuracy.

25.3 Enterprise Knowledge Graphs

Knowledge graphs are used in enterprises as a core product or service, as an asset that powers various internal applications, or as an enabler for data and knowledge exchange across organizations and departments. In this section, we will introduce such enterprise knowledge graphs from different perspectives. First, we will cover the knowledge graphs that serve for data and knowledge integration inside an enterprise; then we will cover applications of data and knowledge integration across enterprises. For this, we will introduce a centralized approach and some decentralized approaches like data spaces, an architectural design principle that provides distributed, sovereignty-oriented data sharing that uses knowledge graph technology in various places.

25.3.1 Data and Knowledge Integration Inside an Enterprise

We have already seen Web search or, more precisely, question-answering tasks and how big tech companies like Amazon, Google, and Microsoft utilize knowledge graphs to power various applications, like virtual assistants. These applications typically work with knowledge across various domains. In this section, we get into

[14] https://link.springer.com/book/10.1007/978-981-19-7596-7

[15] There are over 360M triples describing book instances in the WebDataCommons crawl from October 2021, which is based on CommonCrawl. http://webdatacommons.org/structureddata/2021-12/stats/schema_org_subsets.html

more domain-specific applications of knowledge graphs. Since covering the entire landscape would require its own book (e.g., Pan et al. 2017), we will cover here several domains that have been particularly active in knowledge graph adoption with some examples. Then following surveys like Hogan et al. (2021) and Pan et al. (2017), we will introduce knowledge graph applications from domains like health, finance, and (social) media.

In the finance domain, data are an important asset for various tasks, such as risk assessment and fraud detection. However, making sense of and using these data is not always straightforward as they are typically highly scattered across different departments and are hidden in databases behind legacy systems. The knowledge graph technology allows the integration of such data, enabling reasoning and data analytics. Such integrated data across different departments in a financial institution may help assess the risk of certain operations. Similarly, entity resolution techniques can be applied to identify fraudulent transactions via checking transaction patterns across similar customers (Deloitte 2020). The Financial Industry Business Ontology (FIBO)[16] is a widespread industrial standard used for knowledge graphs in the financial domain. In addition, financial institutions like Wells Fargo[17] and Capital One[18] are known to use knowledge graph technology for various tasks, such as question answering, recommendations, risk analyses, semantic search across customers and products, and more.

In the health domain, the application of semantic technologies has a long history. In fact, one of the first applications of Linked Open Data was Bio2RDF (Belleau et al. 2008), a project that publishes various health- and life-science-related datasets as linked data. More than a decade later, knowledge graphs have been used extensively in various research activities in the health domain, including drug development, clinical trials, and, most recently, COVID-19 research (Rajabi and Kafaie 2022).

Knowledge graphs can find application fields not only in academic research but also in application-specific enterprise settings in the health domain. A recent concrete example is the Elsevier Health Knowledge Graph,[19] which helps clinicians with the diagnosis and treatment of their patients. The knowledge graph provides integrated, succinct, trustworthy, and accurate knowledge collected across various materials, such as medical textbooks and journals, as well as various patient guidelines. The knowledge graph is mainly used for question answering, for example, for a query like "ketamine dosage for pain" (Kamdar et al. 2021).

Finally, in the (social) media domain, knowledge graphs are typically used to link people, their friends, interests, and other entities on the Web and the media around them. A well-known application in traditional media is linked knowledge graphs published and utilized by the BBC. Currently, they offer a search interface over a

[16] https://spec.edmcouncil.org/fibo/

[17] https://www.youtube.com/watch?v=25UIgiiYqsE

[18] https://www.youtube.com/watch?v=ABN2377ER_A

[19] https://www.elsevier.com/about/partnerships/research-in-healthcare-collaborations

knowledge graph under the name of BBC Things.[20] An example application of such a knowledge graph is content recommendations, for example, recommending audio and video content related to an article a user is reading on the BBC Web site. Another application in the media domain is the IMDB Knowledge Graph, which integrates knowledge about movies, actors, box office numbers, and more.[21] The knowledge graph is used to power various applications, for example, providing movie recommendations (e.g., given a movie search by a user, what the closest movies are in the IMDB database to the searched movie). A common application of knowledge graphs for the social media field is the Facebook Social Graph,[22] where people, their interests, businesses, groups, and locations are integrated.

The landscape of knowledge graph applications in enterprises is quite large, and we have only given a small portion in domains where knowledge graphs are particularly well adopted. Further application areas and examples can be found in various sources, such as Hogan et al. (2021), Pan et al. (2017), and Noy et al. (2019).

25.3.2 Data and Knowledge Exchange in Enterprise Networks

So far, we have covered the application of knowledge graphs in enterprises where data and knowledge primarily originate from a single organization. In many cases, however, multiple organizations can integrate their data to increase the value of various applications. Such knowledge graphs facilitate data and knowledge exchange in these networks. An enterprise network, in this sense, can be implemented in two ways: centralized and decentralized.

25.3.2.1 A Centralized Solution

In the centralized scenario, various organizations integrate their data into a knowledge graph that is controlled by a specific central body. This large, centralized knowledge graph can then be accessed via various applications, including the ones from individual organizations. This allows knowledge and data exchange between different parties through a central intermediary. However, individual organizations do not have much control over how much data should be shared with whom beyond what they send to the central knowledge graph. A prominent example of a centralized enterprise network is the German Tourism Knowledge Graph.[23] Sixteen regional marketing organizations from Germany push their relevant data into a

[20] https://www.bbc.co.uk/things/about

[21] https://www.youtube.com/watch?v=TFNLt2UyvQI

[22] https://developers.facebook.com/docs/graph-api

[23] https://open-data-germany.org/datenbestand-such-widget/

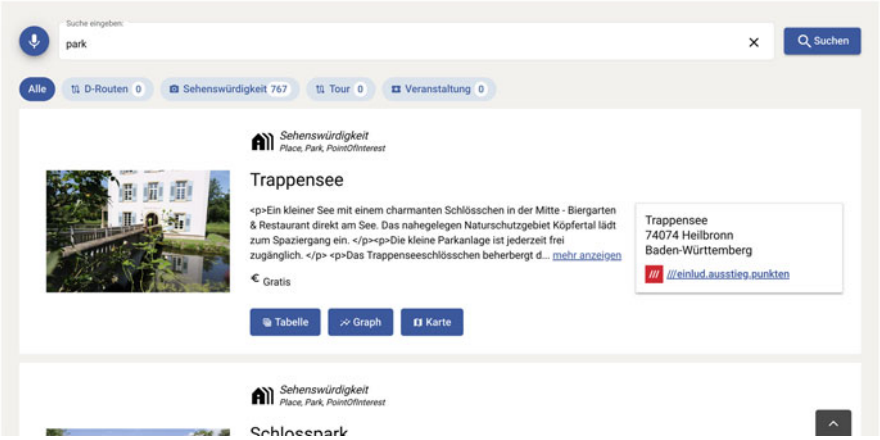


Fig. 25.8 A Web-based application powered by the German Tourism Knowledge Graph

knowledge graph curated by the German National Tourism Board (GNTB). GNTB fully maintains the knowledge graph, and it provides the required infrastructure to facilitate the data and knowledge exchange. The applications are built on top of this central knowledge graph. Figure 25.8 shows the user interface of a Web-based application for browsing the knowledge graph.

25.3.2.2 Decentralized Solutions

In the decentralized scenario, data and knowledge exchanges are instead done in a federated and peer-to-peer fashion. Data ownership stays fully on the owner's side, offered to consumers via connectors. The owners can create policies regarding what other parties are allowed and not allowed to do with the data they offer. The data are accessed and integrated in a federated manner by the consumers. The core technical principle contains the unambiguous identification of data assets, trust management, policies, and, as enablers to all of these, the semantic self-descriptions of different assets. These self-descriptions created with semantic technologies are published in federated catalogs to enable discovery by consumers and applications. The catalogs of asset descriptions can be modeled as knowledge graphs to host semantic self-descriptions. A currently popular way to implement decentralized enterprise networks is the **data space** approach.

Data ownership, privacy, and protection are becoming increasingly important topics for sharing data on open infrastructures such as the Web or using them for cloud computing and storage. In the following, we will discuss some initiatives that are trying to provide solutions for these issues. They are not only relevant for individual users and their wish for data sovereignty but also for larger organizations that process and store data for their customers. Many organizations collect and store data where there is a significant duty to protect the privacy of clients and other

consumers. Think of health care, insurance, legal services, and more. Data sovereignty means that these data holders can safeguard user data and ensure that it is used only in accordance with strictly defined rules.

There is also the aspect of the *value* of these data. Many data providers currently give their data away or use them as currency in exchange for services and other considerations from large data platforms. For others, data-sharing hurdles create a drag on efficiency or barriers to entry into a market for smaller players. Providing means for proper data sharing can generate a booming economy around data value chains and prevent parasitic economic models where data are collected from users without a clear value for them. These aspects will grow in importance, given trends like the Internet of Things or the cyber-physical space (see later sections).

Solid (derived from "social linked data") is "a proposed set of conventions and tools for building decentralized social applications based on Linked Data principles. Solid is modular and extensible, and it relies as much as possible on existing W3C standards and protocols."[24] It was announced in 2018 by Tim Berners-Lee, followed by a start-up called *Inrupt*, to provide infrastructure to Solid. The general idea is that users own and host their data, and applications (ala Facebook) access it to the extent that users allow them. The cornerstones of Solid are:[25]

- *True data ownership*: users should have the freedom to choose where their data stay and who is allowed to access them.
- *Modular design*: because applications are decoupled from the data they produce, users will be able to avoid vendor lock-in and can seamlessly switch between apps and personal data storage servers without losing any data or social connection.
- *Reusing existing data*: developers will be able to easily innovate by creating new apps or improving current apps, all while reusing existing data that were created by other apps.

Obviously, these points bother all the Web sites that have used user data as the core of their business model. They want to own this user data and do not want an autonomous user.

The **International Data Spaces (IDS)** initiative aims to generate a safe domain-independent data space, allowing small and large enterprises to manage their data.[26] The core is a *reference architecture model*[27] developed by the Fraunhofer-Gesellschaft. The reference architecture model has the following layers; see Bader et al. (2020):

[24] https://solid.mit.edu/

[25] Taken from https://solid.mit.edu/

[26] https://internationaldataspaces.org/why/data-sovereignty/?gclid=EAIaIQobChMIneXOi5P%2D%2DQIVTf7VCh3BjwxpEAAYASAAEgKYfPD_BwE#mia

[27] https://www.fraunhofer.de/content/dam/zv/en/fields-of-research/industrial-data-space/IDS-Reference-Architecture-Model.pdf

- The *business layer* defines and categorizes for the participants their different roles and the interaction patterns they can make use of. The roles include data owner, data provider, data consumer, data user, intermediary, etc.
- The *functional layer* deals with trust, security, data sovereignty, data ecosystem, interoperability, and data markets.
- The *process layer* defines the various processes and their interactions that can be run over the platform.
- The *information layer* (Bader et al. 2020) is a straightforward RDFS/OWL ontology, and SHACL shapes to provide self-descriptions defining the schema of the digital contents that are exchanged over this platform.[28] SHACL should be used for validation and SPARQL for retrieving self-descriptions. The self-descriptions should be stored in federated catalogs.
- The system layer comprises three major elements: connectors, the broker, and an app store:
  - *Connectors* can be internal or external. Internal connectors should provide secure access to a data service provided via the platform (i.e., via the App Store). An external connector manages the exchange of data between participants of the international data space platform.
  - A *broker* manages the process model of the platform and is implemented as a connector.
  - The *app store* provides data apps for processing data in the framework of the IDS platform.

GAIA-X[29] is an effort to build a reliable and scalable data infrastructure for European private and public data providers. It is a top-down-driven initiative mainly by Germany and France with the support of the European Commission. We sketch here the core architectural elements:[30]

- A provider operates resources in a GAIA-X ecosystem and offers them as a service. It provides a service instance, together with self-description and policies.
- A federator helps connect service providers and service consumers.
- A consumer consumes service instances in the GAIA-X ecosystem to provide offers for end users.

An important principle of GAIA-X is *machine-readable self-description* of architectural assets and participants used for:

- The discovery of an asset in a catalog
- The evaluation, selection, and integration of service instances and data assets
- Enforcement, validation, and trust monitoring, and
- Negotiations concerning assets and participants

[28] https://international-data-spaces-association.github.io/InformationModel/docs/index.html

[29] https://gaia-x.eu/

[30] https://www.heise.de/downloads/18/2/9/0/6/1/1/7/gaia-x-technical-architecture.pdf and https://gaia-x.eu/wp-content/uploads/2022/06/Gaia-x-Architecture-Document-22.04-Release.pdf

GAIA-X uses a *triple-based data model forming a graph* for machine-readable self-description. The formalization of a schema for GAIA-X self-descriptions should become an RDFS/OWL ontology, which is, however, a work in progress.[31] The document GAIA-X—Architecture Document—22.04 Release[32] sketches an information model (i.e., self-descriptions) referring to the usage of W3C standards and JSON-LD. "Self-Descriptions are W3C Verifiable Presentations in the JSON-LD format. Self-Description consists of a list of verifiable Credentials. Verifiable Credentials themselves contain a list of Claims assertions about Entities expressed in the RDF data model." Still, there are no statements of the expressive power of the used language nor an explanation of how you verify RDF/OWL specifications based on their usage of the open-world assumption.

GAIA-X presents dozens of use cases from different domains (GAIA-X 2021). Here, we introduce one from the energy domain. *Enershare*[33] is a project that aims to create a common European energy data space. The aim of the project is to enable a data-driven energy ecosystem that brings various stakeholders from the energy and other related sectors (e-mobility, water, health) together with information and communication technology (ICT) stakeholders. The data space could foster the secure sharing and exchange of data assets and their usage via various applications and services. A GAIA-X-compliant architecture will enable federated and secure data sharing in business-to-business (B2B) and business-to-consumer (B2C) transactions. There are many use cases that can work with such a common European data space, like better "energy roaming" for electric vehicles. Many electric vehicle drivers struggle with charging their cars when they go away from their home environment, even more when they travel internationally. They need multiple subscriptions or mobile applications for different charging networks. The prices across charging networks are not comparable due to a lack of transparency, and the lack of knowledge about whether charging stations are available and functional at any given time makes it challenging to plan longer journeys. An application on top of a common European energy data space where charging point operators, electric vehicle producers, and utility providers are participating can solve these problems as it would enable these parties to share data in a secure and transparent way.

IDS also presents several use cases (IDSA 2022), like the one from Deutsche Telekom. The *Data Intelligence Hub* (DIH)[34] is an overarching use case across many domains that provides a data marketplace. Consumers and providers can publish the self-descriptions of their data in the Data Intelligence Hub, and the marketplace brokers data exchange between consumers and providers. This brokerage follows the core principles of the data space approach. Data providers have full control over their data, and the exchange is done directly between consumers and providers. The marketplace provides a secure environment that brings these parties

[31] https://www.researchgate.net/publication/348767747_GAIA-X_and_IDS

[32] https://gaia-x.eu/wp-content/uploads/2022/06/Gaia-x-Architecture-Document-22.04-Release.pdf

[33] https://enershare.eu/

[34] https://portal.dih.telekom.net/marketplace/

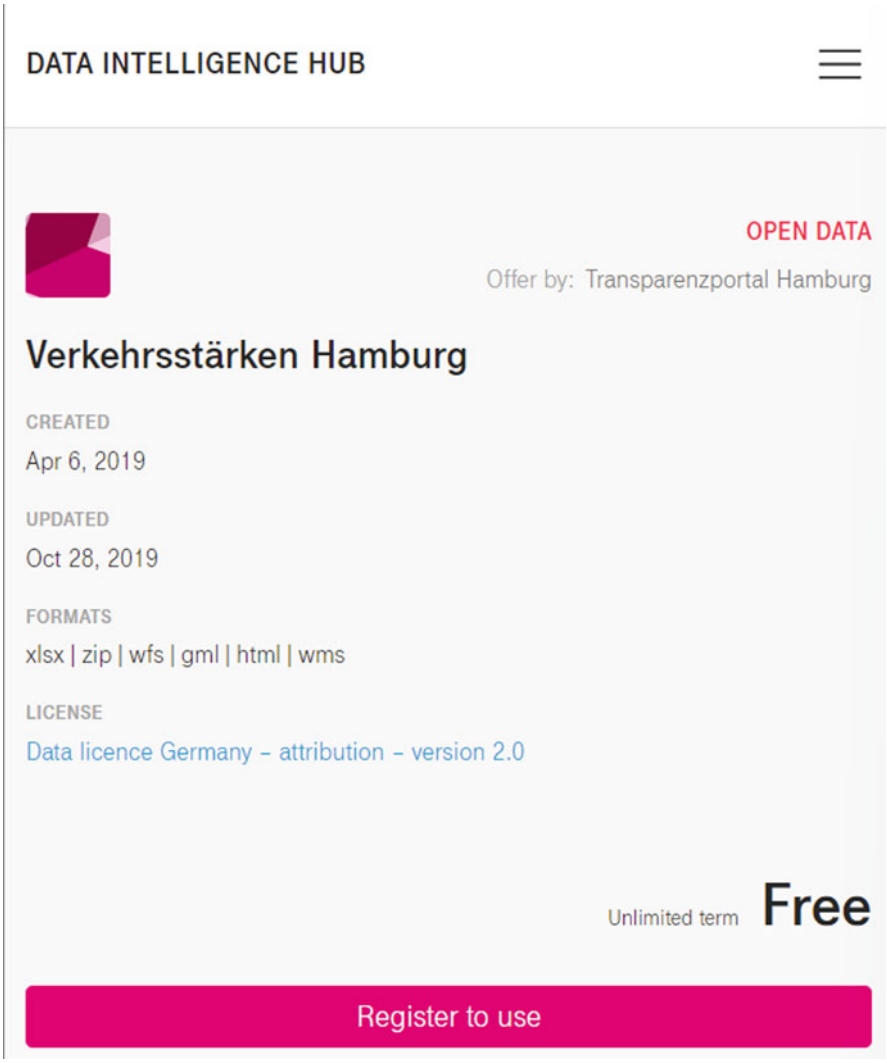


Fig. 25.9 A data offer published on the Data Intelligence Hub

together and brokers data sharing, between organizations and various applications that can make use of the data to provide services like analytics. Figure 25.9 shows a screenshot from the DIH marketplace. It shows an offer from "Transparenzportal Hamburg"[35] for an asset about traffic data. The consumers can see various details about the data asset, such as the creation and last update dates, available formats,

[35] https://transparenz.hamburg.de/

license, and pricing information, as well as the availability duration of the offer. In this case, the offer is available for free for an unlimited term for the registered users of the Data Intelligence Hub.

25.4 Cyber-Physical Systems and Explainable AI

Cyber-physical systems operate at the intersection of the cyber and physical worlds, meaning they contain physical components such as electrical and mechanical actuators and sensors that interact with the physical world and cyber components that deal with software and data that interact with the cyber world. These two worlds can also have an impact on each other; for example, the data collected based on an interaction with the physical world via sensors can affect the decisions made in the cyber world, which then again can have an impact on future actions in the physical world.

Cyber-physical systems are an important type of new-generation manufacturing systems as they enable smarter manufacturing processes, for example, predicting errors in production devices and automatically scheduling maintenance based on previous patterns. The major challenge in cyber-physical systems is integration. Integration can be considered in two categories: low-level integration, which deals with the integration within a system between the physical and cyber components, and high-level integration between different cyber-physical systems (Jirkovsky et al. 2017). The low-level integration challenge occurs due to the multidisciplinary modeling that requires different perspectives of the system, such as mechanical, electrical, and software perspectives. The models from these three perspectives may be heterogeneous in terms of the meaning of the concepts and their relationships, constraints, and so on. For example, a conveyor belt is seen as a system consisting of a belt, motor, and roller from the mechanical perspective, whereas the software perspective also models a motor control unit. Moreover, the properties and their values for a component, like a belt, can be different in different knowledge graphs that are integrated to create a final model of a cyber-physical system (Grangel-Gonzalez et al. 2018). This model can be used for knowledge exchange and various tasks such as simulation and testing. The high-level integration challenge arises due to the different data models and interfaces used by different manufacturers of cyber-physical systems. When these systems need to work together in the physical world, such as a conveyor belt and a welding machine, their descriptions must be aligned to integrate them into the cyber world. Like low-level integration, knowledge graphs support such alignments.

Aside from integration, knowledge graphs are also used to explain the behavior of complex cyber-physical systems. One example is the anomaly explanation on smart energy grids, a cyber-physical system that has multiple stakeholders. A smart energy grid may contain a photovoltaic plant for electricity production, an electric vehicle charging station, and the users of such a station. A smart electric grid may reduce the amount of electricity produced on the grid because of a series of calculations

depending on a set of factors (e.g., weather conditions). Such anomalies in the grid may occur without any explanation of the cause (e.g., a machine learning model decides on the production reduction transparently). The knowledge about anomalies and their causes represented in a knowledge graph can be used to create explanations for such changes in the smart grid system (Aryan 2021).

Our final example for the application of knowledge graphs in cyber-physical systems stems from automated driving. Automated driving vehicles make use of many real-time and historical data collected from various sources, including a variety of sources, to make decisions in different traffic situations. The data collected from such sources are used to train machine learning models; however, typically, the semantics of the relationships between these entities are not well represented. An approach presented by Halilaj et al. (2022) uses knowledge graphs to improve the performance of ML models supporting automated driving. The architecture of the system consists of three major layers:

- *The data layer* is the lowest layer that contains datasets in different modalities, such as images from different sensors, geolocation data from GPS, and geographical data from maps. The raw data may be already annotated to some extent but not necessarily in a semantically rich way.
- The *knowledge layer* is on top of the data layer and contains various knowledge graphs. The local knowledge graphs contain semantically enriched data from each data source in the data layer. They are then integrated into a global knowledge graph. The annotations from the data layer may be beneficial during the semantic enrichment process in terms of mapping to different domain ontologies.
- Finally, the *application layer* contains various modules to support automated driving that consume the global knowledge graph from the knowledge layer, for example, to predict the behavior of an immediate object while driving.

Orthogonal to these three layers, various other external sources can also be integrated into the system, such as weather data and traffic rules. More examples of how the created knowledge graph can be used for different automated driving tasks can be found in Halilaj et al. (2022).

We will encounter an exponentially growing merger of the physical and virtual worlds. Physical agents act in the virtual world, and virtual agents act in the physical world. It may even become difficult to distinguish these two worlds. We already see autonomous cars, autonomous robots, and drones. Hardware and bioware will merge more and more to the point where it will be even harder to tell whether an agent is a human or a physical agent guided by a virtual intelligence. Every object of the physical world must go cyber and interact there to remain visible and existing. Each cyber agent must know as much as possible about the physical world to prevent acting like *a bull in a china shop*.

"Explainable Artificial Intelligence"[36] is the future of AI and requires semantics. Current AI systems are often probabilistic (based on machine learning). Delivering a

[36] https://de.wikipedia.org/wiki/Explainable_Artificial_Intelligence

more transparent AI experience for users is both of high research interest and of practical significance. *Making AI easily understood by humans requires semantic interpretation*, which could prevent accidents such as the incident in March 2018, where Elaine Herzberg was the first victim of a fully autonomous driving car.

25.5 Summary

With the motto "The proof of pudding is in the eating," the real power of knowledge graphs manifests itself in the applications. There is already a plethora of applications in the last decade, and they can be covered from various dimensions. We chose to delve into four categories that have a substantial impact on everyday life, namely, search engines, virtual assistants, enterprise knowledge graphs, and cyber-physical systems.

Improving the scalability of search and information retrieval is one of the initial motivations of semantic technologies and natural knowledge graphs. The knowledge graph technology is at the core of the migration of search engines toward becoming query-answering engines as they contain semantically enriched data about different domains. This allows search engines to directly answer user queries without redirecting them to other Web sites.

Similarly, virtual assistants are also powered by knowledge graphs. No matter how well the speech recognition and natural language understanding modules work, a virtual assistant is pretty much useless without access to vast knowledge, which is well provided by knowledge graph technology.

Knowledge graphs are also an important enabler when it comes to knowledge and data exchange within or across enterprises. On one side, organizations from many domains use knowledge graphs to integrate data that are produced in various places across their departments but typically hidden behind legacy systems and databases. These knowledge graphs are used for different applications that benefit from the added value generated by semantically described, connected data. On the other side, knowledge graphs enable data sharing across enterprises in a centralized or decentralized manner. In the centralized approach, the knowledge graph integrates data from different organizations and is curated by a central intermediary. In the decentralized approach, data ownership stays with individual providers, and a federated brokerage system (e.g., a catalog of metadata) enables data and knowledge exchange in a peer-to-peer fashion. Consumers and providers find each other via semantically annotated self-descriptions of their assets. We presented a recent approach to decentralized enterprise networks called data spaces.

Finally, we presented exciting new developments in cyber-physical systems around manufacturing, energy, and automated driving. The applications presented there are all backed by various industrial stakeholders, which indicates that knowledge graph technology will continue to be the driver of applications that have a large impact on our lives. The automated driving vehicle that has the knowledge of traffic

rules and the behavior of the parties in the traffic we dreamt of in Part I may not be too far away.

References

Aryan PR (2021) Knowledge graph for explainable cyber physical systems: a case study in smart energy grids. In: Proceedings of the 9th workshop on modeling and simulation of cyber-physical energy systems, CEUR-WS, vol 3005

Bader S, Pullmann J, Mader C, Tramp S, Quix C, Müller AW, Akyürek H, Böckmann M, Imbusch BT, Lipp J et al (2020) The international data spaces information model – an ontology for sovereign exchange of digital content. In: International semantic web conference. Springer, pp 176–192

Belleau F, Nolin MA, Tourigny N, Rigault P, Morissette J (2008) Bio2rdf: towards a mashup to build bioinformatics knowledge systems. J Biomed Inform 41(5):706–716

Deloitte Netherlands (2020) Knowledge graphs for financial services. Technical report, Deloitte. https://www2.deloitte.com/content/dam/Deloitte/nl/Documents/risk/deloitte-nl-risk-knowledge-graphs-financial-services.pdf

GAIA-X Data Space Business Committee (2021) Position paper: Consolidated version for industry verticals. Technical report, GAIA-X. https://www.gaia-x.eu/wp-content/uploads/files/2021-08/Gaia-XDSBC_PositionPaper.pdf

Grangel-Gonzalez I, Halilaj L, Vidal ME, Rana O, Lohmann S, Auer S, Müller AW (2018) Knowledge graphs for semantically integrating cyber-physical systems. In: Database and expert systems applications: 29th international conference, DEXA 2018, Regensburg, Germany, September 3–6, 2018, Proceedings, Part I 29, Springer, pp 184–199

Halilaj L, Luettin J, Henson C, Monka S (2022) Knowledge graphs for automated driving. In: 2022 IEEE Fifth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), IEEE, pp 98–105

Hogan A, Blomqvist E, Cochez M, d'Amato C, Melo G, Gutierrez C, Kirrane S, Gayo JEL, Navigli R, Neumaier S et al (2021) Knowledge graphs. ACM Comput Surv 54(4):1–37

Ilyas IF, Rekatsinas T, Konda V, Pound J, Qi X, Soliman M (2022) SAGA: a platform for continuous construction and serving of knowledge at scale. In: Proceedings of the 2022 international conference on management of data, pp 2259–2272

International Data Spaces Association (2022) Data spaces overview. Technical report, IDSA. https://internationaldataspaces.org/wp-content/uploads/dlm_uploads/220812_Use-Case-Bro2022_35-MB.pdf

Jirkovsky V, Obitko M, Marik V (2017) Understanding data heterogeneity in the context of cyber-physical systems integration. IEEE Trans Ind Inform 13(2):660–667

Kamdar MR, Dowling W, Carroll M, Fitzgerald C, Pal S, Ross S, Scranton K, Henke D, Samarasinghe M (2021) A healthcare knowledge graph-based approach to enable focused clinical search. In: Proceedings of the ISWC 2021: posters, demos and industry tracks: from novel ideas to industrial practice co-located with 20th international semantic web conference (ISWC 2021)

Noy N, Gao Y, Jain A, Narayanan A, Patterson A, Taylor J (2019) Industry-scale knowledge graphs: lessons and challenges: five diverse technology companies show how it's done. Queue 17(2):48–75

Pan JZ, Vetere G, Gomez-Perez JM, Wu H (eds) (2017) Exploiting linked data and knowledge graphs in large organisations. Springer

Pellegrino MA, Santoro M, Scarano V, Spagnuolo C (2021) Automatic skill generation for knowledge graph question answering. In: The Semantic Web: ESWC 2021 Satellite Events: Virtual Event, June 6–10, 2021, Revised Selected Papers 18, Springer, pp 38–43

Radford A, Narasimhan K, Salimans T, Sutskever I, et al. (2018) Improving language understanding by generative pre-training. Technical report. https://www.cs.ubc.ca/amuham01/LING530/papers/radford2018improving.pdf

Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I, et al (2019) Language models are unsupervised multitask learners. Technical report. https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe

Rajabi E, Kafaie S (2022) Knowledge graphs and explainable AI in healthcare. Information 13(10): 459

Şimşek U, Fensel D (2018) Intent generation for goal-oriented dialogue systems based on schema.org annotations. arXiv preprint arXiv:180701292

Zou X (2020) A survey on the application of knowledge graph. J Phys Conf Ser 1487:012016