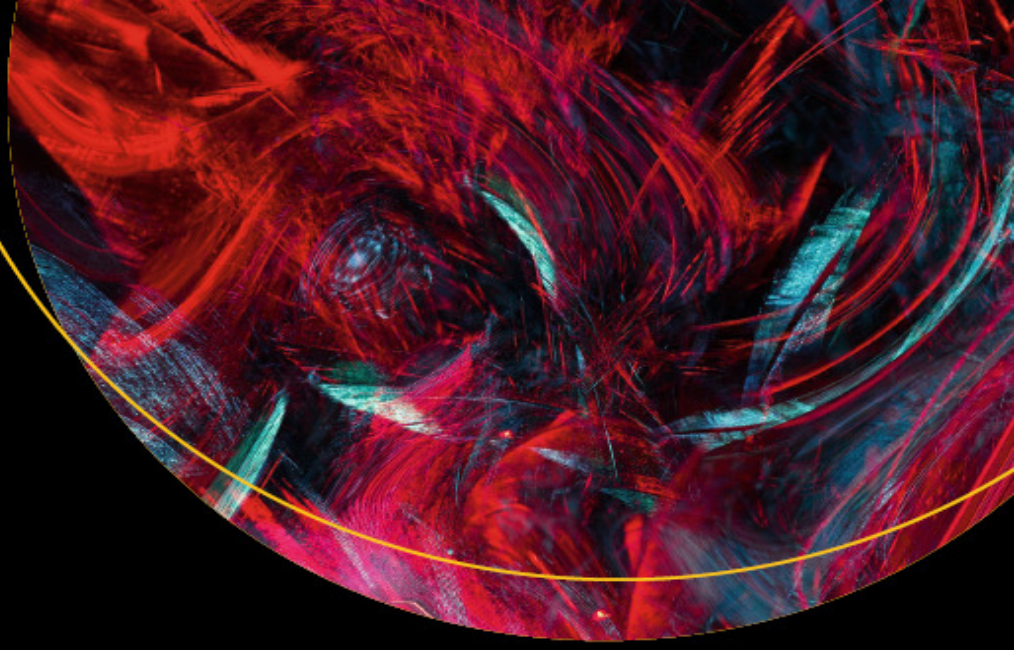


Designing Semantic Knowledge Graphs

A Guide to Utilizing Semantic Web Technologies for Agile Data Management

Michael DeBellis

Foreword by Dr Robert T. Neches

Apress®

Michael DeBellis

Designing Semantic Knowledge Graphs
A Guide to Utilizing Semantic Web Technologies for Agile Data Management

Foreword by Dr Robert T. Neches

Apress®

Michael DeBellis
San Francisco, CA, USA

ISBN 979-8-8688-1822-6 e-ISBN 979-8-8688-1823-3
https://doi.org/10.1007/979-8-8688-1823-3

© Michael DeBellis 2026

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. Unless otherwise expressly agreed by the Publisher in writing, the Publisher does not authorize use or reproduction of all or any part of this work in any manner for the purpose of training, fine-tuning, evaluating, or developing artificial intelligence models, technologies, or systems. In accordance with Article 4(3) of the EU Directive 2019/790 on copyright and related rights in the Digital Single Market, the Publisher expressly reserves all rights with respect to this work under the text and data mining exception of the directive.

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.

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.comwww.springeronline.com, or visit . Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.
The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

To Dr. Diane Zacher, whose support made this book possible.

Foreword

This book is coming out at an especially important time for both research and practice of systems incorporating AI. From a research standpoint, it brings attention to critical parts of the field that have been largely overshadowed by publicity for LLMs. This has gotten to a point where many falsely see “AI” and “chatbots” as synonyms. From the standpoint of anyone interested in getting practical value out of AI, this book is even more important. Its coverage is comprehensive. It teaches the basic coding languages and techniques needed to build the connections between AI and conventional system components. It shows how to control and manage large, data-intensive systems. And it also shows how those techniques enable building software engineering best practices into systems, making them far easier to develop and maintain.

Organizations today possess more data than ever before. Extracting full value remains remarkably difficult. Data is distributed across systems, described using inconsistent terminology, governed by different assumptions, and often separated from the knowledge needed to interpret it. New developments in artificial intelligence have only exacerbated the difficulties. Systems can process enormous amounts of information, but sorely need reliable ways to represent meaning, context, relationships, provenance, and constraints.

Designing Semantic Knowledge Graphs provides key parts of the solution. The book does an excellent job of showing how to use them. Combining graph-based data with formal models of meaning allows systems to move beyond merely *storing* information. It facilitates understanding *how that information fits together*. It further enables better sharing of that understanding not only within software implementing system components, but also among the many human stakeholders: developers, managers, subject matter experts, customers, users. The tools help integrate heterogeneous data, make assumptions explicit, support reasoning and validation, and provide a foundation for systems that must be both intelligent and trustworthy.

Despite their potential, semantic technologies are often presented in one of two unsatisfactory ways. Often, they are treated primarily as an abstract research topic, with an off-putting emphasis on formalism at the expense of the practical realities of building production systems. Alternatively, they are reduced to tutorials on particular languages or tools, without explaining how those technologies fit into a broader engineering process. *Designing Semantic Knowledge Graphs* bridges these gaps.

Michael DeBellis approaches ontology and knowledge-graph development as an engineering discipline. The book begins with the ideas and standards that make Semantic Knowledge Graphs possible, but it does not stop with definitions or syntax. It follows the complete development life cycle: understanding requirements, developing conceptual and design models, implementing those models in code, integrating data, validating results, and addressing governance and security. It then places these techniques within larger architectural contexts, including data products, Domain-Driven Design, Data Mesh, Data Fabric, enterprise integration, machine learning, retrieval-augmented generation, and large language models.

This book spans requirements, to models, to code, all within the context of large-scale system development. You can find other discussions of OWL, RDF, SPARQL, or SHACL in isolation. However, only here is that coupled with explanation of how the technologies work together and how early decisions in requirements analysis affect modeling, implementation, validation, and governance. This breadth is the book's greatest strength.

Just as important, it explains why particular approaches are useful. Readers learn not only how to construct a class hierarchy, write a SPARQL query, or define a SHACL constraint, but also when those techniques are appropriate, what problems they solve, and what tradeoffs they introduce. The emphasis throughout is on *making sound engineering decisions* rather than mechanically applying standards.

The book is not an encyclopedia of every graph technology, artificial-intelligence technique, or commercial product. It is not a

comprehensive textbook on formal logic or Description Logic. It provides what you need to understand how logic supports ontology modeling and includes additional material for those who want to explore further. It does not pretend ontologies and Semantic Knowledge Graphs replace relational databases, event streams, machine learning systems, or large language models. It shows how modern systems require several of these approaches—and how to make them work together.

Thus, the book concentrates on the distinctive role that semantic models can play within such systems. Ontologies provide explicit, reusable representations of domain knowledge. RDF supplies a flexible graph-based data model. OWL supports logical inference. SHACL provides validation and operational constraints. SPARQL makes it possible to query and transform graph data. Provenance and metadata vocabularies help organizations understand where information came from, how it has changed, and whether it can be trusted. Together, these provide a semantic foundation other components can leverage.

Readers who are new to semantic technologies will benefit from reading the book sequentially. The first part introduces the fundamental concepts across the lifecycle from requirements through modeling, implementation, governance, and security. The second shows how those support broader enterprise architectures and emerging applications.

More experienced readers may use the book selectively. An ontology engineer might begin with the discussions of SHACL, implementation, or enterprise architecture. A data architect might focus first on data products, bounded contexts, Data Mesh, and Data Fabric. A developer working with generative AI might turn directly to the integration of knowledge graphs, vector search, retrieval-augmented generation, and large language models, returning to the earlier chapters as particular modeling questions arise.

An information architect will find the discussion of how to use models such as glossaries and taxonomies as foundations for ontologies to be most useful. In all these examples, Michael draws on best

practices from leaders in the field such as Dehghani's work on Data Mesh and Jessica Talisman's concept of the Ontology Pipeline from Information Architecture to Ontologies.

In all cases, this is a book to use rather than merely read. There are detailed examples of all concepts including OWL models, SHACL constraints, SWRL rules, and Python code. These examples are all included, along with additional detailed material, on the book's GitHub site available via an open-source license. Readers should work through the examples, examine the accompanying OWL ontologies and SHACL models, run the SPARQL queries and updates, and experiment with the Python code. The examples can then be modified to reflect your own terminology, data sources, constraints, and use cases. Semantic modeling becomes most valuable when it is connected to an actual domain and tested against real questions.

The timing of this book is especially appropriate because systems used for consequential decisions require grounding, validation, provenance, and governance. Modern language models amply demonstrate why knowledge representation is a key adjunct. Fluency is *not* the same as accuracy. Statistical association is *not* the same as explicit meaning.

Michael explains this complex subject in the conversational tone of a skilled mentor sharing experience with a friend. He combines technical depth with an understanding of how real projects succeed and fail. The result is a comprehensive and practical guide to building modern systems capable of extracting maximum value from data.

For many readers, this may well be the only book they need to begin designing and implementing Semantic Knowledge Graphs. I am deeply impressed with it.

Robert T. Neches

Former DARPA Program Manager and Director of the Distributed Scalable Systems Division at the Information Sciences Institute (ISI)

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Acknowledgments

I owe my deepest thanks to Bob Neches, who provided invaluable feedback on drafts of this book. Bob has been a mentor since he reviewed the first research paper I ever published and has always been a model of how to do excellent research. His work on knowledge representation, including NIKL, Loom, and the DARPA Knowledge Sharing initiative, helped establish many of the ideas that made Semantic Web languages possible. This book has benefited enormously from his guidance, insight, and example.

I am also grateful to Franz Inc. for its technical support of my research and for providing early access to AllegroGraph's LLM capabilities. In particular, I want to thank Jans Aasman, CEO of Franz Inc., and Craig Norvell, Vice President of Global Sales and Marketing, for their encouragement and support.

Thanks to Ellie Young, CEO of Common Action, and Cara Foss Arellano, Founder of https://www.DaanMatch.com, for giving me access to substantial real-world data that helped demonstrate the practical value of Semantic Knowledge Graphs. Their work helped ground several of the ideas in this book in the kinds of complex, messy, and meaningful data problems that organizations face in the real world. Thanks to Dr. Nivedita Dutta for providing domain expertise on the first graph retrieval-augmented generation (RAG) system I developed. Thanks also to Dr. Biswanath Dutta, with whom I have enjoyed collaborating on our CODO project, which used knowledge graphs to analyze data from the COVID-19 pandemic in India. He also did a technical review of the book and provided very valuable feedback.

Thanks to the UC Berkeley Data Discovery program, which provided me with several excellent interns to work on various research and client projects. Many of those interns continued working on projects even after they had completed the work necessary for course credit, and their help was invaluable in the research and development that led to this book. In particular, I thank Tejas Jyothi, George Gino, Aadarsh

Balaji, and Jacob Gino. Thanks to Elise for inspiration and for believing in me when even I didn't. Finally, thanks to Oriana and Eden, whose curiosity, resilience, and warmth have been a source of joy, perspective, and hope throughout the long work of writing this book.

Table of Contents

[Chapter 1: Setting the Stage](#)

[What Is a Semantic Knowledge Graph (SKG)?](#)

[Why Semantic Knowledge Graphs?](#)

[Conventions](#)

[Terminology](#)

[Accessing Examples and Getting Help](#)

[Roadmap to the Chapters Ahead](#)

[Part 1: Foundations of Semantic Knowledge Graph Design](#)

[Part 2: Enterprise Architectures and Organizational Patterns](#)

Part 1: Foundations of Semantic Knowledge Graph Design

[Chapter 2: What Is the Semantic Web](#)

[The Original Semantic Web Vision](#)

[The Reality: The Modern Semantic Web Stack](#)

[Vision vs. Reality](#)

[The Foundation Layers: Transport, Unique Identifiers, and Basic Graphs](#)

[RDF Schema Vocabulary and Annotation](#)

[Conceptual and Design Models](#)

[Reasoners and Query Languages](#)

[Labelled Property Graphs](#)

[Linked Data: A Distributed Web of Data](#)

[Notable Linked Data Sources](#)

[Vocabularies](#)

Accessing Knowledge Graphs from Programming Languages
Graph Database Clients
Knowledge Graph Programming Libraries
Applications
Summary
Chapter 3: Core Technologies
RDF/RDFS Semantic Networks
Internationalized Resource Identifier (IRI)
Prefixes
IRIs and Labels
Defining Labels
Resources May Have More Than One IRI
Labelled Property Graphs (LPGs)
RDF-star and Property Graphs
Other Labelled Property Graph Databases
Choosing Between Semantic Knowledge Graphs and Property Graphs
The Strengths of Semantic Knowledge Graphs
When to Consider Property Graphs
The Future of Property and Semantic Knowledge Graphs
SPARQL: The Query Language for RDF Graphs
Structure of a SPARQL Query
Basic SPARQL Queries
SPARQL Queries to Generate Labels
Examples of Linked Data with a SPARQL Endpoint Using Wikidata

Advanced SPARQL Queries
Conceptual Modeling with OWL
Basic Set Theory
Basic Logic
Merging Set Theory with Logic
Quantification Operators for Defining Sets
The Three Kinds of OWL Classes
Relations and OWL Properties
Three Kinds of Properties
Property Characteristics
Semantic Web Rule Language (SWRL)
Defining Classes with Axioms
The Open World Assumption (OWA)
Design Modeling with the Shapes Constraint Language (SHACL)
What Is SHACL, Really?
The SHACL Editor in Protégé
Understanding a SHACL Validation Report
Viewing the SHACL RDF Graph
OWL: A Peek Under the Hood
Summary
Chapter 4: The Data Life Cycle
Overview of Life-Cycle Activities
Agile and the Data Life Cycle
Finding Your First Project

Summary
Chapter 5: Infrastructure and Standards
Ontology Modeling
Reusable Vocabularies
Semantic Knowledge Graph Databases
SPARQL Engine
Knowledge Graph Visualization
Data Ingestion
Version Control and Collaboration
Loading Ontology Files from GitHub
Programming Language Library and Graph Database Client
Ontology Documentation Generation
Automatic Testing Tools
Refactoring Tools
Anti-pattern Analysis
Large Language Model (LLM)
Summary
Chapter 6: Requirements Acquisition
The Role of a System Narrative in Requirements Definition
Example Narrative (Enterprise Data Catalog Project)
Key Elements of a Narrative
Epics in Agile
Modeling Individual Requirements
Stories

Use Cases
Competency Questions
Selecting the Appropriate Requirements Model
Analyzing Requirements Documents
Define Metadata
Low-Fidelity Prototyping
Summary
Chapter 7: Conceptual Modeling
Design Principles for Conceptual Models
Conceptual Modeling and Design Modeling
Buckets
Multiple Inheritance
Basic Modeling Decisions
Modeling the World vs. Modeling a Domain
Class or Instance?
What Type of Property?
Subclass or Collection/Role?
Semantics or Efficiency?
Ontology Reuse
Technical Vocabularies
Domain Vocabularies
Upper Models
Importing Vocabularies
Ontology Patterns

Agent (Party)
N-ary Relations (Property Graphs)
The Collection Pattern: When to Reuse vs. Differentiate Properties
Ordered Value Partition
The Measurement Pattern
Using Sample Instances in Conceptual Modeling
Early Error Detection with the Reasoner
Bottom-up Design
WebProtégé
Using LLMs to Define Conceptual Models
Using an LLM As a Subject Matter Expert
Using an LLM to Define Conceptual Entities
Using an LLM to Define Test Data Instances
Using Generated Test Data
Differences in Coping with OWL Reasoner Errors
Summary
Chapter 8: Design Modeling: From Meaning to Robustness
What Is a Design Model?
A Minimal Graph-Theory Primer for Semantic Knowledge Graphs
Graph-Theoretic Implications for TBox Modeling
Graph-Theoretic Implications for ABox Modeling
The Relationship of Graph-Theoretic Issues to Tool Functionality
The Role of Semantic Knowledge Graphs in Real Architectures
Design Constraints from Distributed Systems (CAP and Scaling)

The CAP Theorem
Vertical and Horizontal Scaling
Sharding and Graph Databases
Design Implications for Semantic Knowledge Graphs
Subgraphs as Design Boundaries
What the Standards Specify (and What They Don’t)
How to Create Subgraphs
Moving Triples Between Subgraphs
Subgraphs As Encapsulation Boundaries
Querying Subgraphs in SPARQL
Data Ingestion As a First-Class Design Activity
Introducing the StreamForge Domain
Relationship to Dehghani’s Daff Example
Data Ingestion Pipeline Example: StreamForge
Overview of the Example
Input Data
Ingestion Run As a First-Class Concept
Constructing the Domain Graph (ABox)
Capturing Provenance (Meta Graph)
Output Files and Graph Separation
Visualizing the Resulting Knowledge Graph
Why This Pattern Matters
Making Design Assumptions Explicit
SHACL As a Derived Design Artifact

Using SPARQL to Operationalize Design (OWL → SHACL)
Datatype Ranges As Validation Constraints
Functional Properties and Cardinality Expectations
Design Implications
Designing Efficient SPARQL Queries
SPARQL Anti-pattern 1: The Cartesian Product
Additional SPARQL Anti-patterns to Be Aware Of
Heuristics for Writing Good SPARQL Queries
ETL, ELT, and Semantic Design
Categories of Data Transformations
Overview of Basic Relational Database Terminology
Row-Level (Record-Level) Transformations
Grouping and Set-Based Transformations
Relational (Multi-dataset) Transformations
Named Entity Recognition (NER)
Data Quality and Validation Transformations
Semantic and Business-Logic Transformations
Ontology Refactoring, Reuse, Validation, and Testing with ROBOT
Automated Refactoring
Ontology Reuse
Validation and Quality Checks
Automated Testing: Overview of the Most Common Technologies
Custom SPARQL Testing
Exposing the Design Model to Distributed Systems

Design Implications
A Note on GraphQL and Semantic Models
Internationalization As a Design Constraint
Language Tags
OWL Profiles and Reasoning Trade-Offs
W3C-Defined OWL Profiles
Product-Specific OWL Profiles
Profiles As a Design-Time Constraint
Summary
Chapter 9: Implementation
Triplestore Graph Database Client
Embedded Libraries
Leading Embedded Libraries
Cloud Landing Zones
Examples of Common Ways to Write Code for Semantic Web Systems
Parameterized SPARQL Functions
Retrieval-Augmented Generation (RAG)
Why Graph RAG Matters
Summary
Chapter 10: Governance, Security, and Enrichment
Governance
Semantic Policy Models for Governance (ODRL)
Data Products and Access Interfaces
Example: A Governance Policy for Accessing a Data Product Port

Governance and Enrichment As Executable Metadata
User-Defined Metadata and Feedback (Collaborative Enrichment)
Integrating Governance, Provenance, and Policy
Internal Customers and Quality Metrics
Security
Enrichment
Summary
Part 2: Enterprise Architectures and Organizational Patterns
Chapter 11: Enterprise Data Products: Microservices for Data
What Is a Data Product?
Integrating Operational and Analytic Data
Data Mesh and Data Fabric: Complementary, Not Competing
Shared Principles and Capabilities
Where They Differ: Governance and Control
The Role of Semantic Knowledge Graphs
A Reference Architecture for Enterprise Data
Event and Messaging Backbone
Service Packaging and Deployment
Orchestration, Scaling, and Discovery
Semantic and Vector Data Stores
Relational Data Stores
Data Pipelines and Workflow Orchestration
Language Models and AI Services
Interface Definition and Contracts

Sidecars: Executing Governance Alongside the Data Product
Why This Architecture Matters
Looking Ahead
Summary
Chapter 12: Semiformal Modeling
SKOS: The Glue Between Informal Models and Ontologies
Glossary
Controlled Vocabulary (Lexical Vocabulary)
Metadata Standards
Taxonomy
Thesaurus
Folksonomy
Example of an Ontology Pipeline from SKOS to OWL
Transforming the Taxonomy into an Ontology
The Root-Level Problem
The Bigger Lesson
Other SKOS Object Properties
Summary
Chapter 13: Data Products and Domain-Driven Design: A Graph RAG Data Catalog
Data Products and Domain-Driven Design: A Graph RAG Data Catalog
Bounded Contexts, Ubiquitous Language, and Explicit Models
Context Maps and Cross-Context Integration
Polysemes and Domain Concepts Across Bounded Contexts

From Bounded Contexts to Data Products

The Role of the Data Catalog

Data Products, Catalogs, and Semantic Knowledge Graphs

The StreamForge Example: Data Products in Context

Visualizing the StreamForge Knowledge Graph

Core Domain Concepts As a Semantic Graph

Data Products Within a Bounded Context

From Conceptual Product to Implementation

Implementing Data Product Ports with REST and OpenAPI

Mapping Ports to OpenAPI Interfaces

Separating Semantics from Interface Structure

Multiple Implementations of a Single Port

Ports As First-Class Architectural Elements

Querying the Data Product Catalog

Grounding Semantic Models in Production Analytics Platforms

Utilizing Large Language Models and SPARQL to Search and Leverage Enterprise Data

Overview of the StreamForge Data Catalog Knowledge Graph

Reuse of Existing Ontologies and Standards

From Data Catalog Knowledge Graph to Graph RAG Catalog Chatbot

Starting Simple: What Is This Catalog For?

Combining LLM and Graph Results: Governance Annotation Example

Provenance and Lineage As First-Class Citizens

Why Textual Annotations Still Matter

More Than One Path Through the Graph

GraphTalker

Benefits for Decision-Makers

Summary

Chapter 14: Semantic Technologies: Industry Applications

Case Study 1: Microsoft Docs—Utilizing Graphwise to Define Semantic Structure at Enterprise Scale

The Semantic Approach: Making Meaning Explicit

Outcomes and Impact

Why This Matters

Case Study 2: A Semantic Layer for an Automotive Enterprise Data Lake

Problem Context

Architectural Approach

Event-Driven Metadata Population

Domain Semantics and Semantic Search

Access Control and Provenance by Design

Implementation and Scale

Why This Matters

Lessons Learned

Case Study 3: PatientGraph Semantic Knowledge Graphs and LLMs for Healthcare Analytics and Decision Support

Context and Problem

Why Semantic Knowledge Graphs Were Needed

Core Capabilities and Architecture

Example Use Case: Clinical Trial Matching

Outcomes and Lessons Learned

Why This Matters

Case Study 4: Avalara—Using Graphwise Neurosymbolic Architecture to Implement GraphRAG

The Graphwise GraphRAG Solution: Separating Knowledge and Communication

Outcomes and Impact

Why This Matters

Case Study 5: Semantic Knowledge Graphs for Essilor Global Supply Chain Risk Management

Context and Problem

Why Semantic Knowledge Graphs Were Needed

Core Capabilities and Architecture

Example Use Case: Supply Chain Risk Assessment

Outcomes and Lessons Learned

Summary

Chapter 15: Conclusion from Models to Systems

Semantic Technologies As Enterprise Infrastructure

Ontologies and Large Language Models Are Not Competitors

From Academic Exercise to Working System

Rejecting False Dichotomies

What This Book Does Differently

The Path Forward

Bibliography

Index

About the Author

Michael DeBellis



is a Semantic Web and AI consultant and researcher with over 40 years of experience in industry and academia. He began designing ontologies in the early 1980s while building expert systems for Accenture and later worked as a researcher at USC's Information Sciences Institute (ISI), where he used Loom—a direct predecessor to the Web Ontology Language (OWL). Throughout his career at firms including Deloitte, Accenture, and ThoughtWorks, he has focused on applying advanced technology to real-world enterprise challenges. His work emphasizes the practical integration of Semantic Knowledge Graphs into complex systems, highlighting the importance of user requirements, legacy data, and enterprise architecture over purely theoretical modeling. He has published and presented internationally on a broad range of topics across multiple domains.

About the Technical Reviewer

Biswanath Dutta



is an Associate Professor at the Indian Statistical Institute (ISI), Bangalore, India, and former Head of its Documentation Research and Training Centre (DRTC). He holds bachelor's degrees in Physics and Library and Information Science (LIS) from Vidyasagar University, a master's degree in LIS from ISI, and a Ph.D. in LIS from the University of Pune, based on research at ISI Bangalore on Semantic Web technologies and e-learning. He was a postdoctoral fellow in the Department of Information Engineering and Computer Science at the University of Trento, Italy, and a research assistant in the Department of Computer Science at Dalhousie University, Canada. His international research activities also include visits and collaborations at the University of Toronto's Interactive Media Lab and the Max Planck Institute for Informatics, Germany. He contributed to the European Commission-funded project LivingKnowledge: Facts, Opinions, and Bias in Time and worked on the India–Trento Programme for Advanced Research (ITPAR II and III). He has also led research projects supported by government agencies and industry. His interests span the Semantic Web, ontologies, knowledge graphs, Linked Data, metadata, knowledge organization, research data management, and applications of AI. He has written more than 100 scholarly publications and four books and supervises research in knowledge representation, semantic technologies, and data management. He is Secretary of the India Chapter of the International Society for Knowledge Organization (ISKO). He serves on the editorial boards of the International Journal of Metadata, Semantics and

Ontologies and BARTOC (Basic Register of Thesauri, Ontologies & Classifications), and on the Editorial Advisory Board of the Journal of Information and Knowledge.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_1

1. Setting the Stage

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

A major shift is underway in how organizations think about data. Old approaches—rigid databases, siloed data—are giving way to dynamic, connected, intelligent models of information. Architectures such as Data Fabrics (Zaidi & Thoo, 2019), Data Mesh (Dehghani, 2022), and Data Centric (McComb, 2019) systems are reshaping the future. Large language models (LLMs) are revolutionizing search both within organizations and for customers. A key enabling technology for this new data-centric world is Semantic Knowledge Graphs. Semantic Knowledge Graphs are a smarter, faster, more adaptable way to organize data—making it easier to build systems that are not only powerful today but also maintainable and extendible for the new architectures enabled by artificial intelligence.

This book offers a practical, industry-tested methodology for designing Semantic Knowledge Graphs. Whether you are a software engineer, business analyst, data architect, taxonomist, or library science professional, this guide will help you design models that are both rigorous and comprehensible. First, we need to set the stage: what are Semantic Knowledge Graphs, and why are they so important right now?

What Is a Semantic Knowledge Graph (SKG)?

At its core, a knowledge graph is simply data organized as a graph and stored in a graph database (Singhal, 2012). A Semantic Knowledge

Graph (SKG) adds a critical layer: it's a knowledge graph built on logical models known as ontologies, providing rich meaning and enabling automated reasoning across data. The idea of ontology dates back to the earliest days of artificial intelligence (AI) (DeBellis & Neches, 2023). An ontology is a formal, logical blueprint of a subject area—defining concepts, relationships, and rules using high-level models that are easy for domain experts to understand and that can be used as the foundation for efficient, scalable data storage and computation.

Many experimental languages for ontologies have come and gone. But in real-world, large-scale systems, one standard has stood the test of time: the Web Ontology Language (OWL), developed by the W3C as part of the Semantic Web initiative. OWL provides the expressive power needed to model complex domains—with the standardization, scalability, reliability, and performance required for serious industry use (W3C OWL Working Group, 2012; Allemang & Hendler, 2011).

In this book, whenever I use the word ontology, I mean an OWL ontology focusing on the schema level: the classes, properties, and logical rules that form the backbone of a Semantic Knowledge Graph. The ontology community calls this the *TBox*, short for "Terminology." In a relational database, this is the schema level. In an object-oriented system, this is the meta-model that defines classes and properties.

When we say knowledge graph, we mean data modeled in graph languages such as RDF, RDF-star, or Labelled Property Graphs—technologies we will explore in the chapters ahead. The ontology community calls this the *ABox*, short for "Assertions." In a relational database, these are rows, and in an object-oriented system, these are instances with values for properties and attributes.

A Semantic Knowledge Graph (SKG) combines these two layers. The ontology provides the logical model that defines the meaning of the terms and relationships (the TBox), while the knowledge graph provides the data that instantiates those definitions (the ABox). In short:

Ontology (logical model) + Knowledge Graph (data) = Semantic Knowledge Graph (SKG).

Figure 1-1, produced with the Gruff visualization tool from the AllegroGraph database, shows a simple example of this structure. The left side illustrates the ontology, the model of classes and properties, while the right side shows corresponding data instances that together form a Semantic Knowledge Graph.

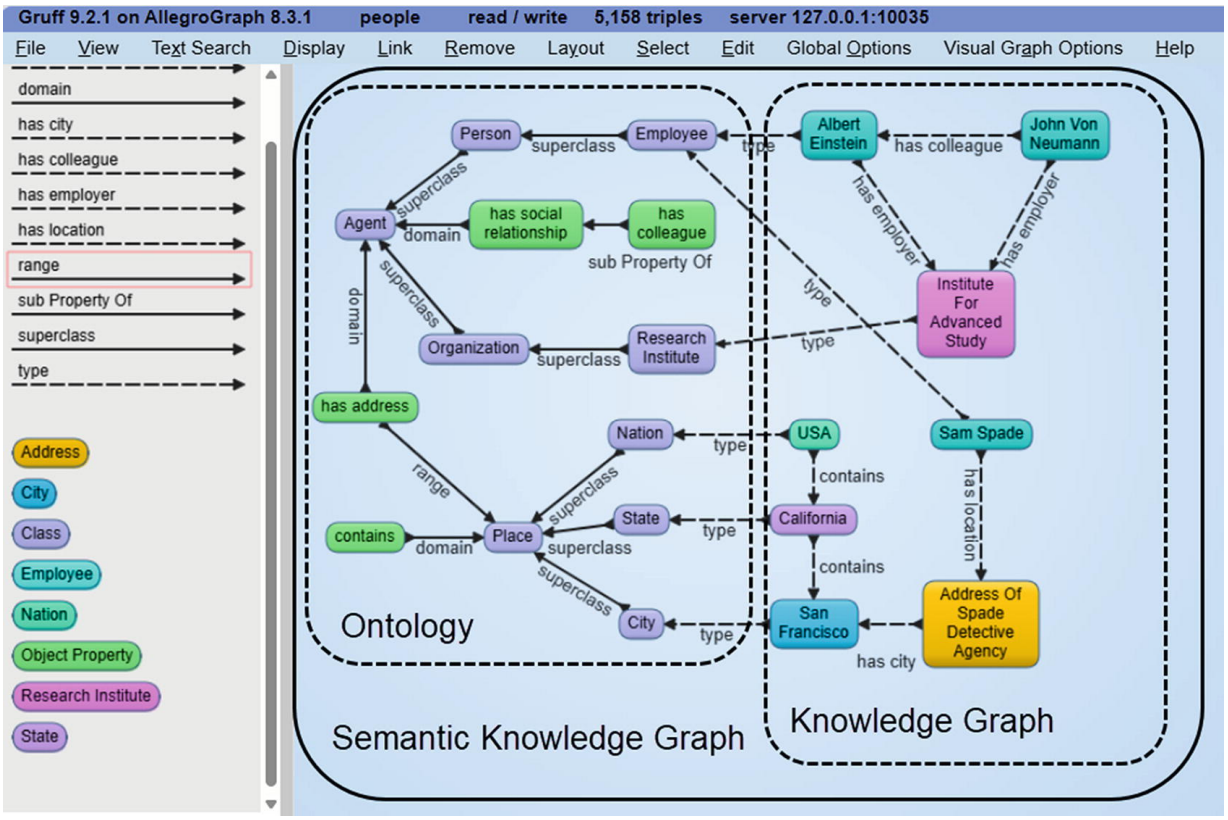


Figure 1-1 Ontology + Knowledge Graph = Semantic Knowledge Graph (SKG)

Gruff provides a key in the left column where it color codes the types of edges such as `contains` and the types of nodes such as `City` and `Class`. In order to accommodate black and white graphics, I've changed edge types so that rather than color coding, the names of the edges are included as labels. I also followed the convention that edges that describe the meta-model or TBox (e.g., `superclass`, `domain`, `range`) are solid lines and edges that are related to the data or ABox

(e.g., `type, contains, has_employer`) are dashed lines. This example describes common, easily understood concepts such as people, organizations, places, and family relations. In this example, the ontology includes classes such as `Agent` with subclasses such as `Person` and `Organization`. Another important class is `Place` with subclasses such as `State` and `City`. We can see that the metadata that describes the model such as `SubClassOf`, `Domain`, `SubPropertyOf` are edges in the ontology graph, just as the actual data containing relations, employer relations, and social relations are nodes and edges in the knowledge graph. One of the most important connectors between the knowledge graph and the ontology is the `rdf:type` property. In the example, `Institute_For_Advanced_Study` is an instance of (i.e., type of) the class `Research_Institute`. `Albert_Einstein` and `John_Von_Neumann` are instances of the `Employee` class and are connected via the `has_colleague` property. The `has_colleague` property is defined in the ontology as a sub-property of `has_social_relationship`. Note that the ontology is also modeled as a graph. This facilitates making changes, even at run time.

There are already many books that describe ontology design. However, they treat the ontology as a self-contained resource rather than as a component of a larger system, thus ignoring the realities of large amounts of data, interaction with large language models, APIs, graphical user interfaces (GUIs) and other real-world requirements. These existing books also ignore the most common kinds of industry systems that a Semantic Knowledge Graph will be a part of, such as Data Catalogs, Data Lakes, and event-based microservices. They therefore miss how requirements of the overall system impact design of the ontology. Such methodologies are appropriate for developers creating ontologies that are meant to be reusable resources in libraries such as the OBO foundry (Jackson et al., 2021). However, they neglect most of the critical issues for ontologies that are components in a larger system.

This book is an attempt to fill this gap.

I emphasize Agile data management because this is an aspect of SKGs that often goes unappreciated. Previous books on ontology design focus on modeling domains and assume the ontology will not thereafter change, except for minor maintenance, implicitly adopting the outmoded waterfall development methodology. Again, this is appropriate for ontologies meant to serve as reusable models, but not for ontologies meant to be part of modern industrial systems. The waterfall model was shown to be inappropriate for software development in the 1980s when Barry Boehm wrote his paper on the Spiral Model (Boehm, 1986). It is even less relevant now that the best software teams focus on continuous integration and continuous deployment (CI/CD). Software development is a knowledge intensive process (Beck, 2004). Developers and customers learn from prototypes and initial releases and make changes to the system which will affect the data model. A waterfall approach for ontology design isn't appropriate for modern industry applications. What's more, it negates one of the greatest advantages that SKGs have over traditional data models: it is far easier to change the data model of an SKG, even at run time, than to change other types of data models.

Why Semantic Knowledge Graphs?

Over the last decade, there has been a revolution in how organizations create, manage, store, and find data. Data Centric, Data Fabrics, and Data Mesh define new data architectures. Data Lakes, Data Lakehouses, knowledge management, and Data Catalogs are the tools used to create and exploit these new architectures. All of these architectures and systems have several requirements in common:

- Control of data is distributed and driven by business units as well as information technology (IT) organizations. This is analogous to the microservices revolution except for data rather than software systems.
- Data can be monetized. Rather than simply being a byproduct of customer transactions as in the past, data such as customer discussion threads, emails, clickstream, and order histories to name

just a few have value, both for the organization that collected it and potentially for resale to others.

- They need to be both secure and open. The Internet not only enabled new ways of doing business, it also enabled new ways to commit crimes. Security is more important than ever. Consumers are now demanding more control of their data such as the right to be forgotten. At the same time, the Internet enabled business-to-business systems that need to interact with each other, crossing organization firewalls.
- They need to support new types of applications. The explosion in AI has enabled new types of systems that assist researchers and engineers with knowledge intensive tasks and act as intelligent assistants to help consumers find bargains and make the most of their purchases.
- They need to be flexible and agile. In an age of continuous integration/continuous deployment (CI/CD), the old data technologies that required taking down a database to make changes to schemas or constructing complex cube architectures that precomputed various ways of "slicing and dicing" data are no longer adequate. Architectures that support changing and even generating schemas at run time are required. Anticipating the way users will need to view and report on data ahead of time is no longer tolerable. In a modern, agile business, we don't know where new opportunities will come from, and we need to be able to query data in unanticipated ways rather than define all possible queries in advance.
- Metadata is essential. Metadata is data about data—it provides the knowledge required by modern architectures and tools to manage and monetize data effectively while ensuring compliance with legal and customer requirements (Zaidi & Thoo, 2019). This is another area where SKGs offer far greater productivity than traditional databases. Built on established metadata standards such as Dublin Core, SKOS, and PROV-O, Semantic Knowledge Graphs can represent both semantic metadata (meaning and relationships) and historical

metadata (provenance and change over time). They also include reasoning engines and query languages that allow this metadata to be analyzed, validated, and transformed automatically.
- Microservices have revolutionized the way large organizations develop and maintain software. Microservices facilitate reuse and rapid, agile development of new business features. This has created demand for the concept of a data product which is to data what a microservice is to business processes. SKGs are ideally suited to define and utilize data products.

These requirements make SKGs a major competitive advantage to manage enterprise data. However, the most important reason why semantic technology matters today is the rise of large language models (LLMs). The impact of LLMs on enterprise software is only beginning to be understood, but one thing is clear: they fundamentally reshape how data architectures function. The new data architectures and applications all have search and indexing as core capabilities and LLMs fundamentally change how both these capabilities can be built and leveraged. The entire sections of books written about enterprise search, data lakes, metadata management, and data catalogs are obsolete because LLMs change the nature of how we interact with data and give us exponentially more powerful and intuitive ways to search.

However, LLMs alone are not enough. Their greatest strengths—scalability, fluency, and adaptability—come with two critical weaknesses: hallucinations and black-box reasoning. Because LLMs do not explicitly represent knowledge, but only capture statistically possible relationships, they have no mechanism to avoid generating and presenting anomalous and incorrect information. Nor do they have means to recognize when they have done so. These confidently presented glitches are called “hallucinations.” LLMs generate responses based on statistical relationships within vast amounts of text, but their internal structures—tensors, matrices, and neural weights—make it nearly impossible to pinpoint where specific knowledge is stored. Even Google DeepMind researchers (Neel Nanda et al., 2023), in experiments probing LLMs, struggled to identify where a model “knows” any specific

fact such as that Michael Jordan is a basketball player. This lack of explainability, referred to as "black-box reasoning," is a fundamental limitation.

SKGs are the antidote to these weaknesses. Unlike LLMs, ontologies and knowledge graphs explicitly encode knowledge in a way that is interpretable by both humans and machines. They provide a structured, logical foundation for organizing and validating information, ensuring that facts remain verifiable rather than probabilistically inferred.

However, semantic technology has weaknesses as well: logic is powerful, but brittle. In OWL and other description-logic-based systems, even a single logical inconsistency can render the entire ontology unusable for reasoning. The inference engine must reject contradictory statements to preserve soundness. This strict requirement for internal consistency gives ontologies their rigor but also makes them sensitive to data from the real world which often has minor errors (e.g., a string value in a property typed as an integer). As Baader et al. note (Baader et al., 2003), description-logic reasoners guarantee correctness only when the underlying knowledge base is consistent.

The future of enterprise AI will not be LLMs vs. SKGs but rather LLMs + SKGs. By combining the scalability and language fluency of LLMs with the precision and explainability of SKGs, we can build systems that augment human intelligence without sacrificing reliability and trustworthiness.

One example of this synergy is the use of knowledge graphs in retrieval-augmented generation (RAG) systems. RAG enhances LLMs by combining structured knowledge retrieval with unstructured language generation, helping mitigate hallucinations while ensuring responses are grounded in curated, real-world data. SKGs play a critical role in these systems by organizing and maintaining metadata about enterprise documents, structured data, and domain-specific knowledge. At the same time, vector embeddings allow LLMs to capture deep semantic relationships, enriching traditional graph-based

reasoning with the nuanced language understanding of neural networks.

Modern knowledge graph platforms increasingly provide native integrations between vector and graph databases, making them feel like a single, unified knowledge base. This is currently one of the most common ways LLMs and SKGs are integrated. However, it is just the beginning—many other hybrid approaches are emerging that will push this synergy even further. In later chapters, I will explore these integrations in greater detail, including new ways to combine explicit knowledge representation with neural models to build more intelligent, explainable AI systems.

This book is intended for readers who want to make practical use of Semantic Knowledge Graph tools and methods. The rest of this chapter provides the information needed to access and understand the tools, data, and examples needed to do so.

Conventions

I'll use the following conventions throughout the book:

- *Italics* will be used for new terminology and emphasis.
- ***Bold italics will be used for strong emphasis.***
- UI elements like tabs, menus, and buttons appear in **bold.**
- References to other sections of the book will appear in **bold.**
- Menus and submenus will be indicated with a "➤" symbol. For example, **Window➤Tabs➤DL Query** means start at the top level menu, choose the **Window** option, then the **Tabs** submenu, and finally the **DL Query** submenu under **Tabs**.
- Code, names of knowledge graph and ontology objects, URLs, IRIs, and file paths are shown in `Constant width font`.
- To enhance readability, I will sometimes refer to ontology entities in normal font, when it is clear from the context that I'm referring to an ontology entity that has been defined and referenced earlier.

Long sections of example code, queries, Turtle, etc., start with:***Listing M-N.*** *Dijkstra's Algorithm*

where M is the chapter number, N is the number of the listing in that chapter, and the text describes the code. All listings that are meant to be used are available on the GitHub site under the directory for that chapter and have titles indicating the listings in that file. Sometimes several listings will be included in one file. A few listings are only meant as examples and aren't meant to be executed. Those listings have no corresponding files on GitHub.

Tip Tips and notes will show up in a box like this.

EXERCISE 1-1: EXAMPLE

Exercises will be shown like this. Sometimes the exercise will just consist of some text. Other times there will be intro text followed by a sequential list of steps. In all cases, the end of an exercise is signified by a black line.

1. First select option **Foo.**
2. Then if you see **Foo Enabled** then do this.

For most exercises, there is additional material that will be useful, especially for those who are less technical. For example, the GUIs for the various tools will be changing, and reprinting a book takes time, so there may be cases where the GUI in the book is no longer consistent with what you see in the latest version of the tool. The files on GitHub should have the most recent versions. In addition, some exercises could benefit from additional screen prints and explanations. For that reason, virtually all the exercises also have additional material on the GitHub site that follow the standard: chapter_number/exercises/name_of_chapter.md. For example, the additional information for Exercise 1-2 can be found at https://github.com/mdebellis/SemanticKG-

Design/blob/main/chapter_1/exercises/EXERCISE_1-2_INSTALLING_PROTEGE.md

---

Terminology

We will define other terms below in the appropriate context. However, one term that's important to clarify right away is *entity*. In ontologies and SKGs, entity is used as a general-purpose term that refers to anything that has a name or identifier in the ontology—classes, individuals, properties, and rules. This usage differs significantly from how the term is used in Entity/Relationship (E/R) models, where an "entity" typically refers to a real-world object or concept, such as employee or department, and is closest to what ontologists would call a class.

While both meanings involve the idea of "something that has identity," the E/R notion of an entity is narrower, excluding relationships and attributes. In contrast, in the ontology world, "entity" is more like a catch-all term—you can think of it as anything that occupies a node or position in the ontology's logical structure.

Why this difference? It is simply due to terminological conventions in different communities. The ontology community needed a general term for "named things in a formal knowledge model," and entity won out—even though it repurposes a term database folks may associate with something more specific. So, if you're coming from a data modeling background and find this confusing at first, you're not alone, and you're not wrong. We're just entering a new neighborhood where the same word has a different meaning—and we'll be using it their way for the rest of the book.

There is a simple ontology called the People Ontology that we will use as our primary example to describe the basic concepts of Semantic Knowledge Graphs. Although the ontology is called the People Ontology, it also includes related concepts such as `Place`, `Animal`, and

`Organization`. These supporting classes help illustrate important modeling patterns, all within a people-centered context.

This and other ontologies, code, and useful documents are available at the book's GitHub repository: https://github.com/mdebellis/SemanticKG-Design. The People Ontology can be found at https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/people_ontology.ttl. I will provide examples from the People Ontology throughout the book. The best way to view most of the examples is with the free open-source Protégé ontology editor from Stanford University.

I will of course define additional terminology in the relevant chapters below.

The next section walks you through how to install and configure Protégé so that you can follow the examples.

Accessing Examples and Getting Help

You can read this book without working through the examples and still get a lot out of it. However, the way it is intended to be used is with many examples of ontologies, knowledge graphs, SPARQL queries, etc., provided on the book's GitHub site. Everything on that site is under an open-source license, so feel free to use the material as a starting point for your own work. The main ontologies are under the directories https://github.com/mdebellis/SemanticKG-Design/tree/main/ontologies and https://github.com/mdebellis/SemanticKG-Design/tree/main/data_product/knowledge_graph. Those directories hold ontologies and knowledge graphs that are used in more than one chapter.

Other files are under the directory corresponding to the chapter they are discussed in. There are a few examples where I placed a few files in more than one place to facilitate making them easy to find, depending on where you are in the book. Some chapters don't have any

examples, so they don't have a directory. I'm hoping this book will be read by people who don't have a programming background as well as those who do because I believe that anyone can learn how to build an SKG. In fact, that is a major difference between the approach described in this book and other ontology methodologies. The existing books emphasize the complexity of ontologies and the need to understand esoteric concepts from philosophy to create "correct" ontologies. This makes practitioners hesitant to build them because they assume they don't have the background. My opinion is just the opposite. Building an ontology is quite easy once you understand the basic concepts because all of us use concepts such as sets, subsets, and logic every day. We just use them so often we aren't even aware of them.

If you have problems with any of the examples or just general questions, go to the discussion section of the book's GitHub site: https://github.com/mdebellis/SemanticKG-Design/discussions. First, look through the existing questions to be sure your question hasn't been asked and answered already. If it hasn't been, then on the left panel of that page, you will see a list of various kinds of discussion categories. Select **Q&A** then click on the green **Create Discussion** button on the right. If you want to launch other types of discussions, simply pick a different category. That will put you into an editor where you can post your question. You can use the GitHub Wiki markup language for things like italics, bold, and code font style, and you can insert screen snapshots or other images. Screen snapshots are very helpful to understand a problem. You can also use the GitHub issue tool to add an issue to the repository.

EXERCISE 1-2: INSTALLING PROTÉGÉ

If you already have Protégé installed, I still recommend that you follow these steps because I have configured Protégé with a few changes that are required for some of the examples. If you have Protégé installed already, make sure it is version 5.6.9 or later. If it is, then go to exercise Loading Plugins and follow the directions to load the needed plugins and to configure the Protégé reasoner.

1. To install Protégé, go to https://protege.stanford.edu/ and click the orange button that says **Download Now**.
2. That should take you to the appropriate download depending on which operating system you use. All the examples in this book were done in Windows; however, any OS that supports Protégé is fine, and the examples will look very similar. On that page, you will also see a link to use WebProtégé. That is also a very useful tool that I will discuss later, but it doesn't currently support a reasoner, which is required for these examples, so it is best to download the desktop version.
3. Once you have downloaded the file and placed it in a directory, double click on the downloaded file to start the installation. It will ask you to confirm a few things. Accepting the defaults is usually the best.
4. Once the download is complete, double click the icon for Protégé to start the editor.
5. If you are going to be using Protégé to build as well as view ontologies, I also encourage you to sign up for the User Support for Protégé and WebProtégé email list. You can do that here: https://protege.stanford.edu/support.

EXERCISE 1-3: LOADING PROTÉGÉ PLUGINS

When you first start Protégé, you should be presented with a pop-up window with all the plugins that are available (see Figure 1-2). Also, you can see this window by using the menu option: **File➤Check for plugins**.

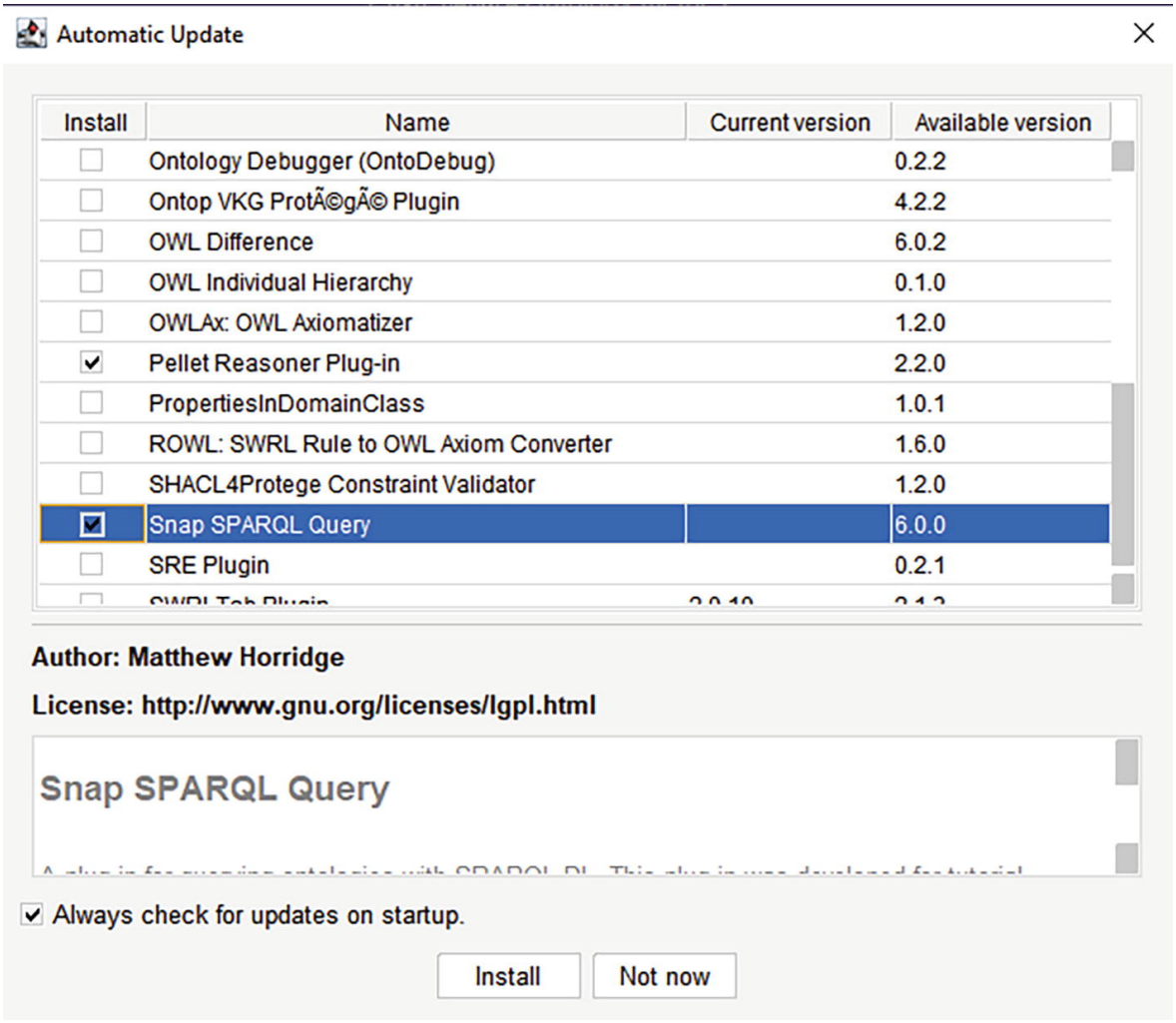


Figure 1-2 Loading Protégé plugins

You will need the following plugins to utilize these examples:

- The Pellet Reasoner
- The Snap SPARQL plugin

Click the check box next to each of these plugins on the plugin window and then select the Install button at the bottom of that window. Be patient, this may take a minute or two. You should shortly get a message that says: **Plugins will be available the next time you start Protégé**. Quit Protégé and restart it so that the plugins are all available.

EXERCISE 1-4: CONFIGURING THE PROTÉGÉ UI

The Protégé UI is highly customizable. It consists of a set of tabs and views. A tab is a big container for one or more (usually several) views. There are several preconfigured tabs that are available by default but that you can also change by adding and deleting views.

To add (or remove) a tab, you use the **Window➤Tabs** option. If you select that option, you will see that several tabs have a check next to them. Those are all the tabs that are currently visible in your version of Protégé. Use Window➤Tabs and select the following tabs:

DL Query. This will allow you to do Description Logic (DL) queries. It will also serve as a place where you can add the Snap SPARQL view described below.

Individuals by Class. This will allow you to view various individuals.

Now that you have all the tabs that you need, you will need to add a view to one of the tabs. To start, select the DL Query tab. Check to see if the Snap SPARQL view is already a sub-tab as in Figure 1-3. If it is, then you are done with this exercise, so move on to Configuring the Reasoner. If it isn't, then this is a good place to add the Snap SPARQL view.

Make sure you are in the **DL Query** tab. Then, select Window➤Views➤Query views➤Snap SPARQL Query. This will give you a blue outline of the new view that you can add to the existing DL Query tab. If you don't see this view, make sure you loaded the Snap SPARQL plugin and restarted Protégé. As you move the outline around, it will change depending on where you put it within the tab. I recommend you position it so it will be a sub-view to the DL Query view as shown in Figure 1-3.

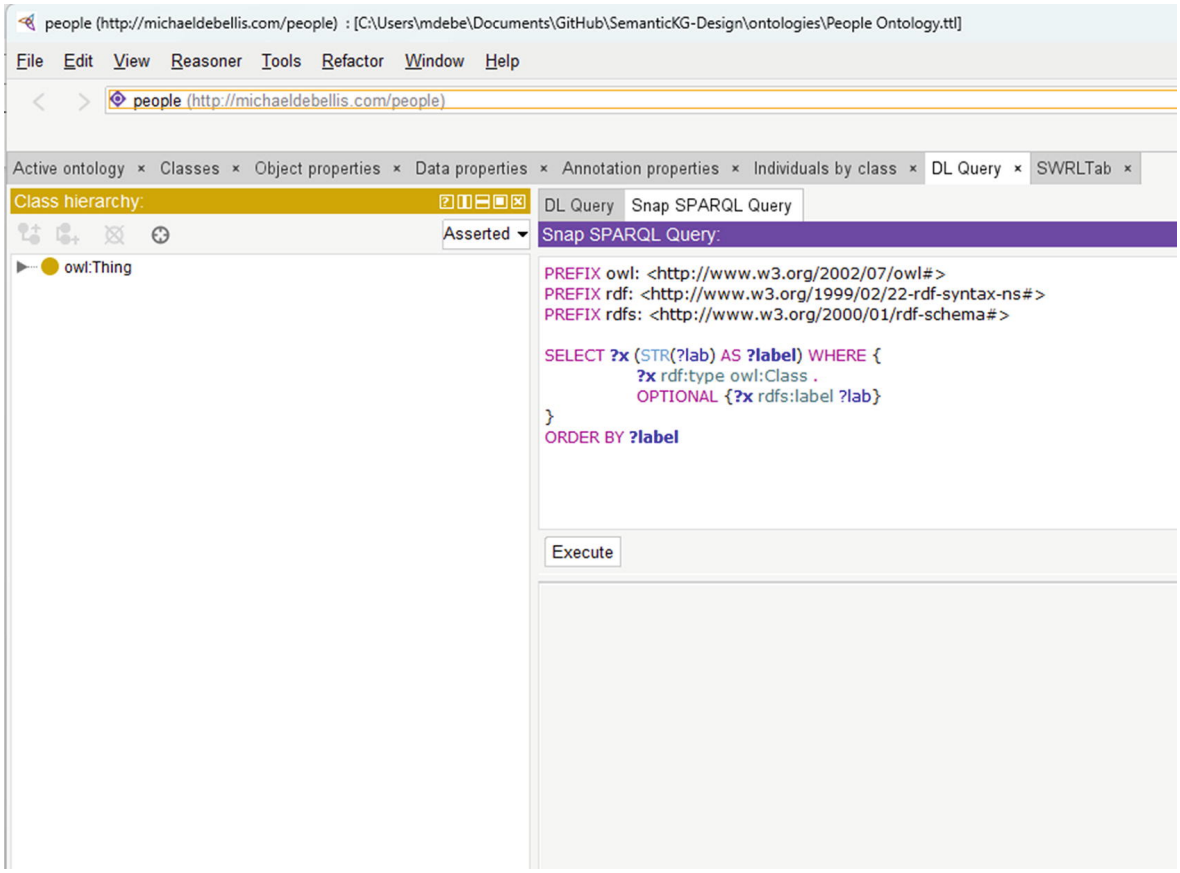


Figure 1-3 The DL Query tab after adding Snap SPARQL

When you have the outline looking like Figure 1-3, click the mouse. It may take you a couple of tries to get it right. It can be a bit confusing the first time. If you add the view in the wrong place, you can just delete it and add it again. Any view in a Protégé tab can be deleted by clicking the "X" in the far right corner of the view. Don't worry about that default query; we'll discuss it below. Also, if you ever accidentally change a tab and want to restore it to its original configuration, go to that tab and select the option **Window➤Reset selected tab to default state.**

EXERCISE 1-5: CONFIGURING THE REASONER

To make sure things are working, we need to load an ontology. Load the People Ontology from the book's GitHub site: https://github.com/mdebellis/SemanticKG-

Design/blob/main/ontologies/people_ontology.ttl. If you know GitHub, you might want to clone the repository. If you don't know GitHub, you can just download files as you need them. In the upper right corner of that page, you will see a download icon where you can download the file. Download the file to your desktop. To load the ontology in Protégé, use File➤Open and navigate to the location where you saved the `people_ontology.ttl` file.

Next, you want to configure the reasoner. Several kinds of inferences the reasoner is capable of are not enabled by default, because for large ontologies, they can take significant time. However, that is not an issue for this simple ontology and for most ontologies we will be using, and you will want to enable all possible inferences in order to understand the full power of the OWL reasoner. To do so, select Reasoner➤Configure. This should bring up a window with two tabs. The default tab: **Displayed Inferences** is the one you want. In that tab, select every box. You may get warnings that a few of the selections may take some time, but again, that is not an issue for the ontologies we'll be using, and in order to understand the power of OWL, it is best to see all the inferences that the reasoner can make. After you have checked every box, select **OK.**

Then, select the Reasoner menu option again. There should be a black dot to the left of **Pellet** indicating it is the selected reasoner. If the black dot is next to a different reasoner, just click on **Pellet** and that will make it the default. The first menu option should be **Reasoner➤Start reasoner**. Use that and start the reasoner. The message in the far right lower corner should say **Reasoner Active** followed by a checked box and Show Inferences. Navigate to the Individuals by class tab and expand the `Agent` class so you can select the `Person` class; you should see several instances of `Person` in the view below the classes. Select `Mary Doe` and your UI should now look like Figure 1-4. Note that all the values highlighted in yellow are not in the definitions from the ontology but rather are inferences from the reasoner based on axioms and rules in the ontology.

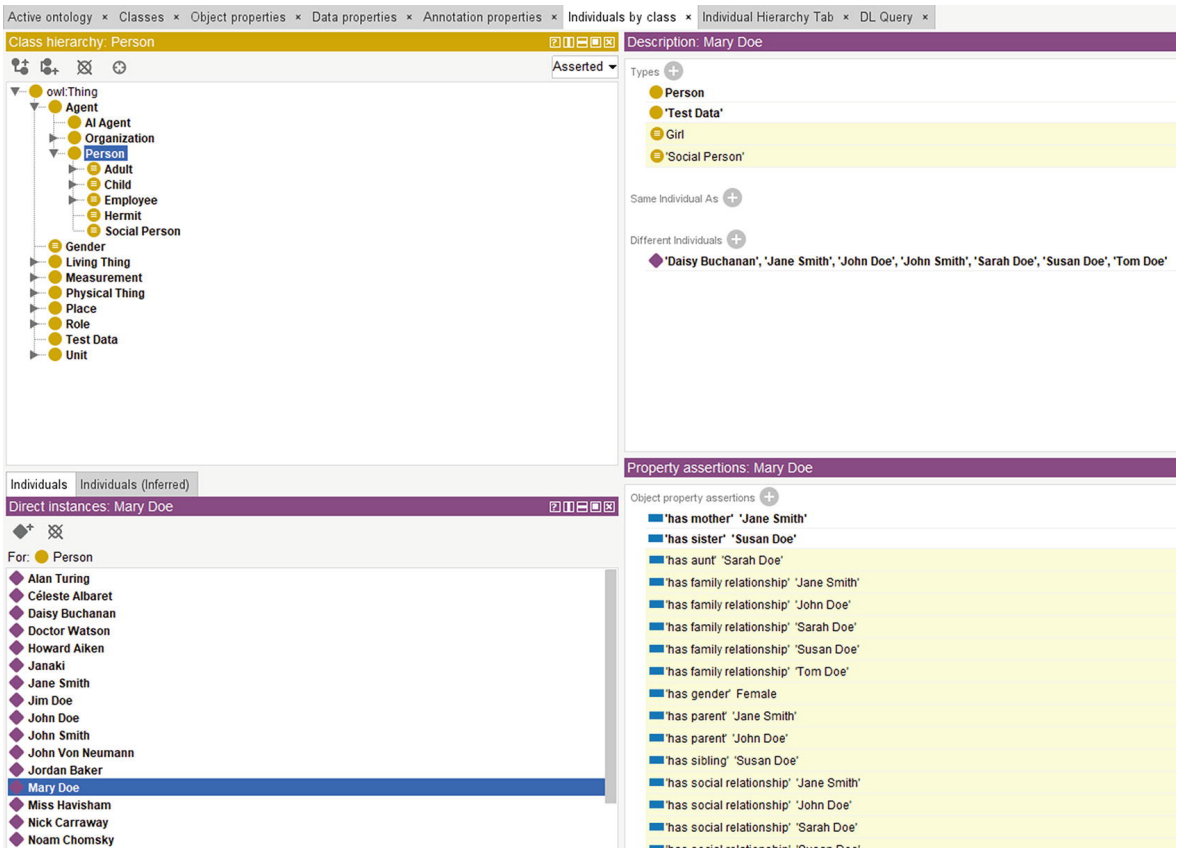


Figure 1-4 The People Ontology in Protégé

Whenever you load an ontology or after you make any changes, that message in the lower right corner will change to **Reasoner state out of sync with active ontology**. This means you need to run the reasoner again. To do so, select **Reasoner➤Synchronize reasoner**. That should get you back to the message that just says **Reasoner Active,** which means the ontology is in sync with the reasoner. Especially for new users, I recommend running the reasoner after every change that could result in new inferences. In practice, this means just about any change except changes to annotation property values which do not affect OWL reasoning. There are three reasons for this:

1. You won't see new inferences that result from your change until you sync the reasoner.
2. The ontologies you work with as a new user are not large so the

reasoner will be very fast.

3. You will find any errors that cause the ontology to be inconsistent right away. This will make it much easier to understand and fix the problem. If there is just one error causing an inconsistency, the reasoner can virtually always tell you exactly what the problem is. One way new users get into trouble is that they make many changes and then run the reasoner. The result can be that because there are several inconsistencies, the reasoner can't give you much help as to what the problems are, so they are much more difficult to fix.

Tip Protégé supports completion. This is very useful. Whenever you are entering the name of any entity, you can type a few characters and then hit <control><space>. If you have typed enough letters to uniquely identify an entity, Protégé will auto-complete the name. If you haven't given enough information, then Protégé will give you a menu with all the entities that start with the letters you've typed. For example, if you type "ha" <control><space>, you will get a list of all the properties that start with "has". This is a great way to reduce work and typing errors. If the ontology has labels defined, you should start typing with a quote. You can still use <control> <space> for completion with labels.

The Search icon. If you want to find an entity but aren't sure of the name, you can use the Search tool. There is a magnifying glass icon in the upper right corner of Protégé. Click on that and start typing, and you will see all the entities that match what you type. You will also see there are many check boxes to configure each search. For example, is it case sensitive, where to search, etc.

Roadmap to the Chapters Ahead

This book is organized into two main parts.

Part 1 introduces the essential tools and techniques—the "hammers, saws, and drills" of semantic modeling. It explains how ontologies, knowledge graphs, and standards such as RDF, OWL, SHACL, and SPARQL provide the foundation for building systems that can represent and reason about data semantics and metadata.

Part 2 turns from the tools to the structures we build with them. Just as a builder or architect must understand the differences between designing a cabin, a bridge, or a skyscraper, this part explores how Semantic Knowledge Graphs (SKGs) support different architectural patterns and real-world applications. It shows how the same foundational technologies can scale from small, domain-specific systems to enterprise-wide knowledge ecosystems.

The following roadmap summarizes how each chapter contributes to this progression, beginning with the origins and core ideas of the Semantic Web and culminating in the integration of semantic and machine learning systems across modern enterprises.

Part 1: Foundations of Semantic Knowledge Graph Design

Chapter 2 provides a high-level overview of the Semantic Web vision that led to Semantic Knowledge Graphs. It traces how the early idea of a web of hypertext links evolved into technologies such as the Web Ontology Language (OWL) and SPARQL, along with pragmatic additions adopted by practitioners, including Labelled Property Graphs and SHACL.

Chapter 3 introduces the core technologies that make up the Semantic Web stack and explains how they fit together to address different modeling and system requirements. It begins at the graph level with Internationalized Resource Identifiers (IRIs), the Resource Description Framework (RDF), and RDF Schema (RDFS). It then examines the Web Ontology Language (OWL) and the Shapes Constraint Language (SHACL), showing how they satisfy complementary needs for logical inference and data validation. The chapter discusses labelled property graphs and why, although they were not part of the original Semantic Web vision, they emerged to

meet practical needs in industry. It also introduces the SPARQL query language as the essential mechanism that ties these components together and shows how SPARQL enables Linked Data and supports querying Semantic Knowledge Graphs.

Chapter 4 presents the data life cycle that guides the rest of the book. Unlike traditional waterfall models, this life cycle is iterative: teams often refine requirements, adjust conceptual and design models, implement code, and enrich data in overlapping cycles. At the same time, the life cycle makes explicit the typical ordering and dependencies among these activities. For example, conceptual models usually precede design models, and meaningful enrichment typically follows initial implementation. This structure helps practitioners understand how semantic systems evolve in practice without implying rigid, lockstep phases.

Chapter 5 examines infrastructure, tools, and standards, sometimes referred to as "Sprint 0" in Agile development. It surveys the open standards and key technologies that enable the life cycle, including modeling environments, triplestores, APIs, reasoning engines, and validation frameworks, and provides concrete examples of each.

Chapter 6 addresses requirements acquisition in detail, offering techniques for eliciting, documenting, and validating semantic requirements. Topics include domain analysis, competency questions, metadata audits, and data profiling. The chapter discusses why competency questions are commonly used by ontologists while also explaining how use cases or Agile stories may be more appropriate for some projects.

Chapter 7 focuses on conceptual modeling, translating requirements into ontological form. The conceptual model—also called the domain or analysis model—is the most abstract type of model and focuses on the business domain without regard to technical implementation. This chapter shows how to design conceptual models using OWL classes, properties, and logical axioms and introduces common ontology design patterns and best practices.

Chapter 8 turns to design modeling, describing how conceptual models defined primarily in OWL can be transformed into design models defined primarily in SHACL. It explains the role of SHACL shapes, how SHACL reasoning differs from OWL reasoning, and how SHACL and SPARQL can be used to define automated validation and regression tests. The chapter also introduces orchestration patterns for ingestion workflows (e.g., scheduling, event triggers, validation gates, and error handling).

Chapter 9 focuses on implementation. It examines common approaches for integrating Semantic Knowledge Graphs with programming languages such as Python and Java, including the use of triplestore client libraries and open-source frameworks such as RDFLib. It also introduces the concept of a cloud landing zone as a foundation for secure, scalable deployment. Finally, it shows how ingestion logic, orchestration definitions, and basic regression tests are implemented and deployed in practice.

Chapter 10 concludes **Part 1** with data governance and security. Governance is presented not as a final step, but as the center of the life cycle, connecting modeling, implementation, operations, and enrichment. The chapter shows how semantic models enable governance by design, supporting access control, lineage, compliance, and controlled enrichment through inference, validation, and automated augmentation. It also introduces key vocabularies used for governance, including Dublin Core for metadata and PROV-O for provenance.

Part 2: Enterprise Architectures and Organizational Patterns

Chapter 11 introduces enterprise data architectures and the trade-offs among centralized, federated, and hybrid models. It explains how SKGs support integration through APIs, ETL processes, and event streams and explores Data Mesh and Data Fabric as complementary approaches to data governance rather than mutually exclusive choices.

Chapter 12 describes semiformal models such as glossaries, folksonomies, thesauri, and taxonomies. It shows how standards like Dublin Core and SKOS can model such semiformal models in a way that makes them stepping stones toward formal ontologies rather than standalone models.

Chapter 13 examines the concept of the data product and the role of Domain-Driven Design (DDD) in structuring microservices. It introduces DDD concepts such as bounded contexts and ubiquitous language and shows how they map naturally onto data products and ontologies.

Chapter 14 presents examples of systems built with Semantic Knowledge Graphs, illustrating how semantic technologies integrate with machine-learning and language-model approaches, including retrieval-augmented generation (RAG). The emphasis is on demonstrating architectural patterns rather than cataloging specific use cases.

Chapter 15 offers the conclusion, summarizing the key ideas from both parts of the book and arguing that the real power of Semantic Knowledge Graphs lies in uniting deductive, logical reasoning with inductive, data-driven learning.

There is a GitHub repository for the book. It provides additional exercises, "cheat sheets," tips, and all ontologies, SHACL, SPARQL, and Python code used in the book, all available via a free open-source license: https://github.com/mdebellis/SemanticKG-Design.

The repository has a directory for each chapter that has code. In addition, the Wiki has many additional materials. The home page for the Wiki is here: https://github.com/mdebellis/SemanticKG-Design/wiki.

One resource that may be especially useful is the hypertext glossary with definitions of all terminology in the book: https://github.com/mdebellis/SemanticKG-Design/wiki/Glossary.

Part 1
Foundations of Semantic Knowledge Graph Design

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_2

2. What Is the Semantic Web

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

The foundation for Semantic Knowledge Graphs is a set of technologies developed by research on knowledge representation done under a vision known as the Semantic Web. The Internet started as a huge network graph formed by URLs and the links to and from them. The original goal of the Semantic Web was to transform the Internet into a semantic graph rather than merely a graph of hypertext links. In hypertext, a link can have countless meanings. It can lead to a login page, to a page that describes the author of the page, to a page on a related topic, etc. The semantics of these links are not explicit. Instead, their meaning is implicit and unavailable to others, buried in the HTML, in code manipulating the pages, and the common sense of the users reading the text. In contrast, the goal of the Semantic Web was to capture the intended meanings (i.e., the semantics) of these nodes and links, along with the metadata required to efficiently utilize them in models that are both intuitive to users and usable by software. The Semantic Web thus provides the foundation for intelligent agents and other new types of systems that go far beyond the conventional Internet.

As often happens in research, the biggest impact was not what the researchers initially expected. The data revolution described in **Chapter 1**, along with a second revolution involving large language models, has created even more demand for semantic technologies that explicitly represent data and metadata and allow organizations to get

the most out of new data architectures and applications. Thus, the modern realization of the Semantic Web is foundational to intelligent enterprise data architectures as well as to the public Internet. In this chapter, we will provide a high level overview of the Semantic Web stack.

The Original Semantic Web Vision

The Semantic Web is a set of standards defined by the World Wide Web Consortium (W3C) that was launched by an article in Scientific American in 2001 (Berners-Lee et al., 2001). In that article, Tim Berners-Lee, James Hendler, and Ora Lassila described how the original goal for the Internet was a web connected by semantic links rather than just hypertext. However, the technology of that time wasn't mature enough to support a Semantic Web, so they compromised on hypertext links. Their paper described research funded by the United States Defense Advanced Research Projects Agency (DARPA) in knowledge representation (Neches et al., FALL 1991) that was paving the way for a truly Semantic Web (DeBellis & Neches, 2023). DARPA is the same agency that funded the research that led to the Internet.

Figure 2-1 shows the original architecture vision for the Semantic Web. This vision is now a reality, although as is often the case, the reality is somewhat different than the original vision.

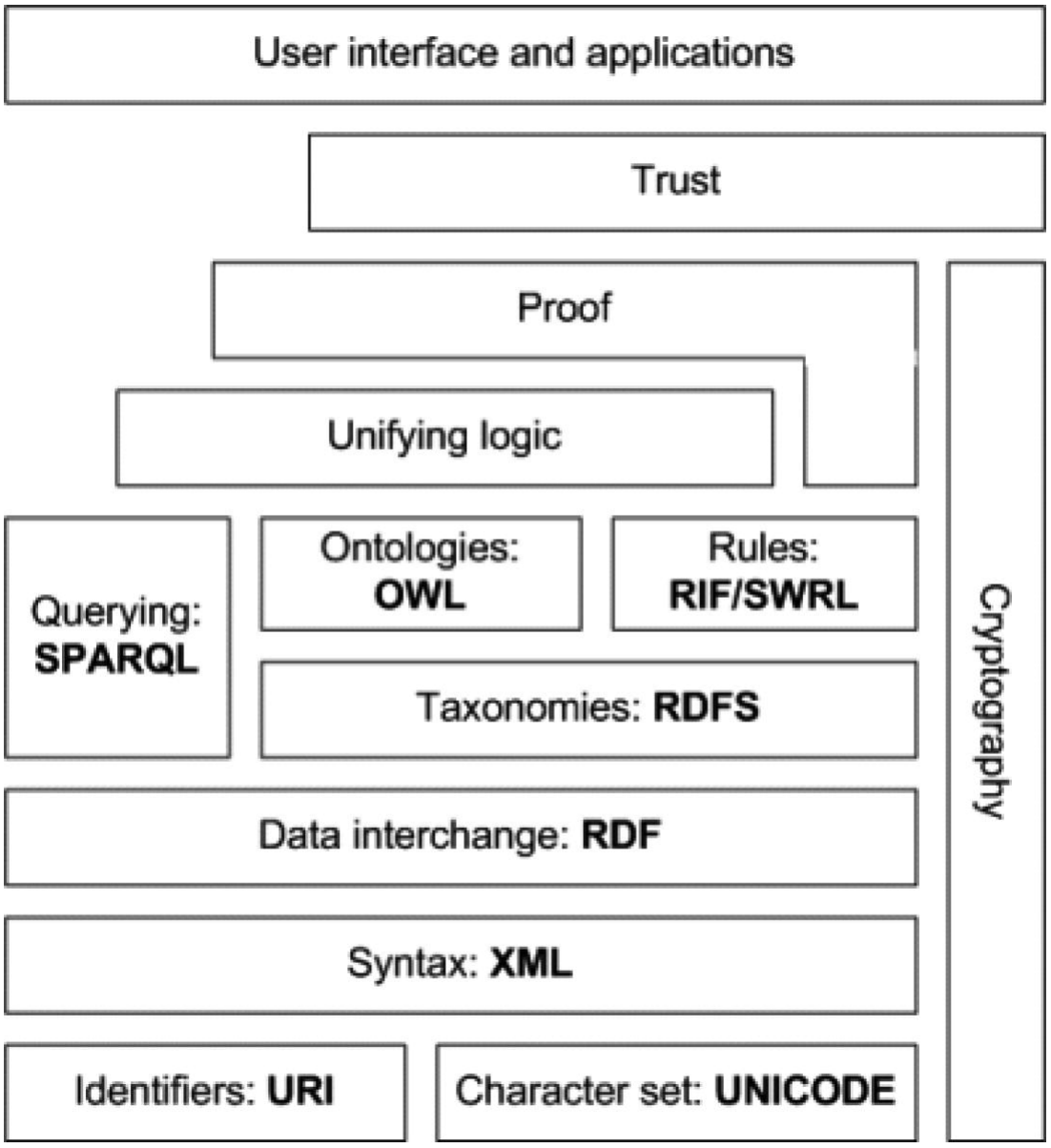


Figure 2-1 The original Semantic Web stack. Source: Wikimedia Commons

The Reality: The Modern Semantic Web Stack

The most important difference between the vision and the reality lies in where the Semantic Web is utilized. The original vision was that the technology would be primarily used to bring meaning to the Internet. Enabling semantics in web pages was expected to help users and autonomous agents navigate more effectively. While this has happened,[1] the driving force for Semantic Web technology has been

behind the firewalls of large organizations: bringing order to the vast amount of data created by the Internet revolution, so that 21st century organizations can utilize it to operate effectively.

Figure 2-2 illustrates the modern Semantic Web stack. We face a bit of a chicken and egg problem here. I want to give an overview of the stack before going into details. In order to do that, I need to use some terminology that will be defined in detail later. So don't worry if you see some unfamiliar terms. This is just a high-level overview to give you the big picture. Any unfamiliar terms will be defined in later chapters. Also, there is a glossary for all terminology at https://github.com/mdebellis/SemanticKG-Design/wiki/Glossary.

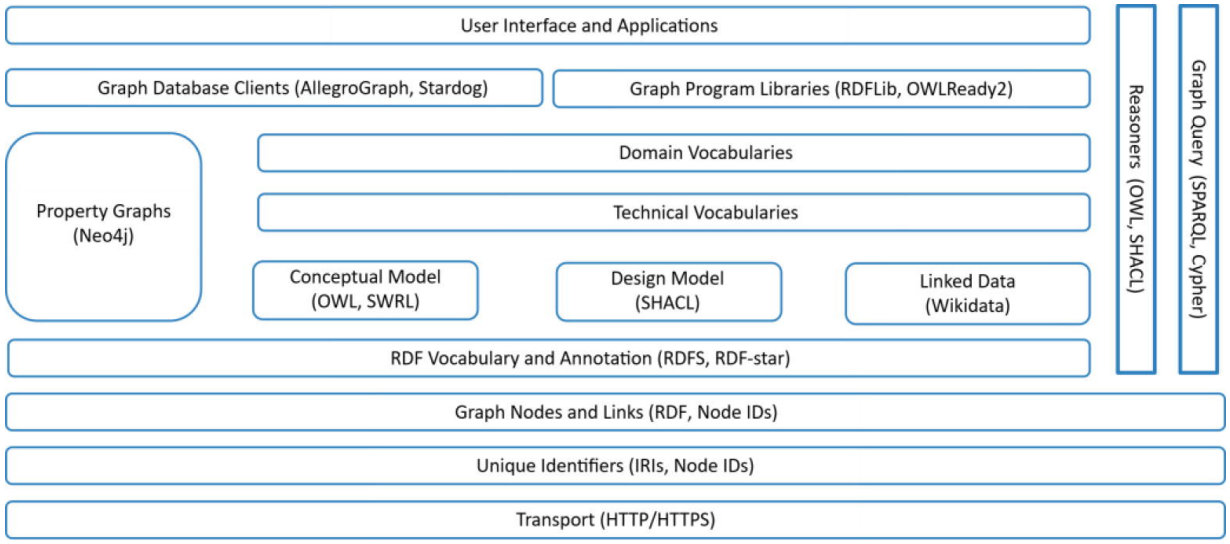


Figure 2-2 The modern Semantic Web (2026)

As I go over the big picture, I'll summarize each part of the stack. In these diagrams, a horizontal layer that is above another layer should be understood as utilizing that lower layer. For example, SHACL and OWL are both defined on RDF/RDFS. Vertical layers such as the SPARQL query language interact with several horizontal layers.

Before describing the different layers, it is worth describing the way the actual Semantic Web differs from the vision described in the

Scientific American paper and the initial Semantic Web stack diagram in Figure 2-1.

Vision vs. Reality

There are two types of differences between the vision and the reality:

1. Components defined in the initial vision that are seldom if ever used in the modern Semantic Web because they were replaced by alternatives or simply never generated much demand from practitioners
2. Components that were not defined in the initial vision stack but were added based on demand from practitioners

The layers that were defined but are seldom used are XML, Trust, and the Rule Interchange Format (RIF). XML has been replaced by JSON or languages such as Turtle. XML was an elegant design because it was so configurable. It was a metalanguage where developers could define their own domain-specific languages. However, this elegance resulted in XML files being verbose. For example, the first common format for saving Semantic Knowledge Graphs was RDF/XML. While RDF/XML was the first official W3C serialization and all fully conformant RDF tools are expected to support it, in practice, it has been largely superseded by more concise formats such as Turtle and JSON-LD, which are now the preferred syntaxes for both human readability and web integration. An example of the verbosity of XML is that ontologies stored in RDF/XML tend to be almost twice as large as the same model stored in Turtle.

The Rule Interchange Format (RIF) was included in the original Semantic Web vision as a framework for enabling different rule languages to share information. It is an official Semantic Web standard, just like OWL, SPARQL, and SHACL, with a formal specification. However, it is not included in the stack diagram because it is seldom used. The main reason for this is that RIF requires developers to

constrain themselves to a common subset of rule features that align with the standard. Since rule languages and engines differ significantly, this limitation proved too restrictive for most use cases. For example, SWRL rules are executed using forward chaining engines, while Prolog is inherently a backward chaining system. These fundamental differences make it difficult to define a meaningful common subset of rule capabilities across multiple systems. Additionally, real-world use cases where two different rule engines need direct logical integration are rare. When rule engines need to share information, they typically do so through APIs or by exchanging data via a shared knowledge base. This approach is far more common than attempting direct logical interoperability via a standardized rule format like RIF. Thus, while the RIF specification exists, it is rarely implemented in practice and will not be discussed further.

Security is critical for any system that exchanges or validates information on the web—but it's also a deep and specialized field, and the standards that emerged for it are primarily not from the W3C. This book focuses on modeling and semantics, so we won't cover security mechanisms in detail. However, here are two key technologies you may encounter:

- *X.509 Certificates*: The backbone of modern security infrastructure, defined by the ITU-T. These certificates associate a public key with a digital identity and are used in systems like HTTPS, TLS, and email encryption. They rely on a Public Key Infrastructure (PKI) and are widely deployed today but are not part of the Semantic Web W3C standards.
- *W3C Verifiable Credentials*: A newer standard from the W3C for expressing credentials in a web-native, semantically rich format using JSON-LD. These are especially relevant for decentralized identity and portable digital credentials, but they are not yet as widely adopted as X.509.

While both approaches support authentication and trust, they are outside the scope of this book. The one exception is the standard ACID

transactions—a foundational database concept ensuring that updates are atomic, consistent, isolated, and durable—which are often required for Semantic Knowledge Graph applications and are provided by most vendors of Semantic Knowledge Graph databases (also known as triplestores). We will discuss those when we discuss data governance and security.

Labelled Property Graphs (LPGs) are an example of a technology that was not in the original Semantic Web vision but has proven to be very useful for knowledge graphs. Hence, the RDF-star standard which is a new addition to the Semantic Web standards for property graphs will be discussed below.

The Foundation Layers: Transport, Unique Identifiers, and Basic Graphs

The bottom layer of the Semantic Web technology stack is built on W3C standards for web transport: HTTP and HTTPS, the latter being a secure extension of the former. Sitting directly on top of this is the IRI layer.

IRI stands for Internationalized Resource Identifier. It plays a crucial role in the Semantic Web by providing a globally unique, standardized way to name things—not just web pages, but also concepts, people, places, ontologies, and relationships. Think of IRIs as the foundation for naming and linking in a globally distributed knowledge graph.

Originally, the Semantic Web used URIs (Uniform Resource Identifiers), which were restricted to ASCII characters. IRIs extended this concept to support the full range of Unicode, allowing identifiers in scripts like Kanji, Arabic, or Devanagari. Today, the W3C recommends using the term IRI, though you'll still see URI used in many documents and tools.

An IRI looks much like a URL. In fact, URLs are one kind of IRI. For example, the IRI for the People Ontology is https://michaeldebellis.com/people.

However, it's important to understand that, unlike a URL, an IRI doesn't have to point to a retrievable document. The actual ontology in this case is stored on GitHub at the URL https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/people_ontology.ttl.

The IRI above is used in the ontology itself to identify the ontology and its contents. This is what makes IRIs so powerful: they give us a way to uniquely identify and refer to resources across the global web of data, whether or not they're retrievable in a browser.

In the Semantic Web, the word resource refers to anything identified by an IRI—a web page, an ontology, a class, a property, or an individual. Everything described in a Semantic Knowledge Graph is a resource.

Building on IRIs, the next layer is the Resource Description Framework (RDF). RDF defines knowledge as a graph of triples in the form Subject ➤ Predicate ➤ Object. The subject and predicate are identified by IRIs, while the object can be an IRI, a literal value (such as a string or number), or an anonymous resource known as a blank node. Because the subject of one triple can be the object of another triple (and vice versa), RDF triples allow us to construct richly connected network graphs of data.

In property graph systems such as Neo4j, the same conceptual role played by IRIs and RDF is handled differently. Each node in a property graph has a unique internal identifier, often called a node ID, and relationships are explicit links between nodes rather than separate triples.

Instead of representing information as subject–predicate–object triples, property graphs attach data directly to nodes and relationships as key–value pairs, known as properties. These identifiers are typically scoped to a single database rather than being globally unique like IRIs, though some implementations allow mapping to external IRIs when interoperability is required.

Conceptually, both approaches describe a network of connected entities, but RDF emphasizes globally linkable identifiers and

declarative statements, while property graphs emphasize efficient traversal and flexible, local labeling of nodes and relationships.

RDF Schema Vocabulary and Annotation

The next layer above this has extensions to RDF: Resource Description Framework Schema (RDFS) and RDF-star. RDFS is simply an RDF vocabulary. It doesn't change the RDF language for defining graphs as triples but simply defines some basic node and edge types such as `rdfs:label` for providing a string to use as a label, `rdfs:Class` to define a class node, `rdfs:subClassOf` to define an edge that relates a sub- and superclass, `rdfs:domain` to define the domain for properties, and `rdfs:range` to define the range of properties. OWL and SHACL, which are discussed in the next section, also are simply vocabularies defined on top of RDF and RDFS.

RDF-star also extends the RDF language. It provides a capability to define properties on properties. This is the knowledge representation capability that distinguishes property graphs from Semantic Knowledge Graphs and RDF/RDFS graphs. RDF-star is discussed in more detail in the section on property graphs below.

Conceptual and Design Models

The next layer includes languages for defining conceptual domain models: the Web Ontology Language (OWL) and design models with the Shapes Constraint Language (SHACL). Just as RDFS is a model defined on top of RDF, OWL and SHACL are models defined on top of RDF/RDFS.

The Shapes Constraint Language (SHACL) is the most significant addition which was not included in the original Semantic Web stack. Shape is a data science term for a data structure that defines the constraints associated with a model rather than the content of that model (as OWL does). This reflects the change in the use of the Semantic Web from a language to bring semantics to the public Internet

to its current focus of helping bring order to organizations' enterprise data.

There is significant overlap between what can be defined in SHACL and the Web Ontology Language (OWL). For example, the domain, range, and cardinality constraints on properties can be defined in both languages. The difference lies not in ***what*** can be defined but in ***how*** the models are used. OWL is a logical language. It is a decidable subset of First-Order Logic,[2] known as Description Logic (DL) and is used to define, validate, and reason over conceptual domain models. OWL is a language for defining the concepts of a model at the highest level of abstraction, before any consideration of technical requirements. SHACL is a less abstract model without logical reasoning. It is the appropriate language for design modeling. When we do conceptual modeling, our only concern is to model the domains, not to consider issues such as how to handle existing data. When we do design modeling that is when we consider issues such as existing data. SHACL is the language used primarily for design modeling as it can tolerate bad data.

The separation between a conceptual model and a design model is useful for teaching and for making the intent behind modeling choices clear, but it should not be mistaken for a strict sequence of steps in a waterfall life cycle. Nor is it the case that all conceptual modeling is done in OWL and all design in SHACL. In real projects, we usually build both kinds of models together and iterate between them. For example, an OWL conceptual model might use multiple inheritance because it reflects how people naturally categorize a domain, but during design, we may transform that structure into single inheritance to satisfy constraints of the implementation platform. These transformations aren't contradictions; they are normal shifts from "understanding ***what*** the domain is" to "understanding ***how*** we can implement the data in the domain effectively." The distinction is about perspectives and tasks, not about rigid phases.

As described in **Chapter 1**, a knowledge graph is any database that models data as a graph rather than with tables or other data structures,

and a Semantic Knowledge Graph is a knowledge graph defined with an OWL ontology as the foundation as shown in Figure 1-1.

Knowledge graph is a broad term. It can refer to graphs that utilize an underlying OWL ontology, but it can also refer to other types of graph data that are not based on an OWL ontology. In actual use, it is very common to integrate data from knowledge graphs that are not based on OWL with Semantic Knowledge Graphs that have a logical OWL foundation. In fact, this is the norm for real-world systems. The vast majority of Semantic Knowledge Graphs built on OWL typically leverage knowledge graphs such as DBpedia and Wikidata, which are based on RDF/RDFS but do not use an OWL logical model.

OWL is a logical language based on set theory and logic. It is the result of research in knowledge representation going back to the 1980s and initiatives such as the DARPA Knowledge Sharing Initiative. An OWL model is called an ontology. Every kind of object in OWL is called an entity. There are four main types of entities:

1. *Classes*: Classes are the main way to model domains. If you are familiar with Object-Oriented Programming, OWL classes will be easy to understand. If you aren't, don't worry; classes are just sets and subclasses are subsets, and although we aren't aware of it, we reason about sets every day, and it's a very natural way to think about the world. The biggest difference between OOP and OWL classes is it is possible to create logical definitions of the necessary and sufficient conditions for an object to be an instance of an OWL class, and those conditions can be used for all sorts of additional validation and reasoning.
2. *Properties*: Properties describe relations between classes as well as relations between classes and datatypes such as strings, integers, and dates.
3. *Individuals*: Individuals are instances of classes. In OOP, many people consider it a good design practice to stick to a mono-hierarchy. That is, the class hierarchy is a tree where each node has only one parent (superclass) node, which means each instance is a

only one parent (superclass) node, which means each instance is a direct instance of only one class. That constraint isn't required in OWL, and there are many ways that using multiple inheritance can create models that are more intuitive and maintainable. Again, if we introspect about how we normally think of the world, we use multiple inheritance all the time.

4. *Rules*: The Semantic Web Rule Language (SWRL) extends the kind of definitions you can create with OWL. There are some conditions that can't be modeled with OWL axioms but can be easily modeled with SWRL rules.

Reasoners and Query Languages

Both OWL and SHACL have their own reasoners. The OWL reasoner makes logical inferences based on the OWL model and the data in the knowledge graph. The SHACL reasoner checks data in the knowledge graph and reports when data violates an integrity constraint. I will go into this in more detail in subsequent chapters, and as I explain how the OWL reasoner works, it will be very clear why OWL is not an appropriate tool for data integrity (i.e., design) constraints. These reasoners are shown as vertical layers in the stack because while they reason over OWL and SHACL models, they can be used both at design time to make sure a model is consistent as well as at run time to do reasoning (OWL) and constraint validation (SHACL) on data created by applications.

SPARQL is a graph query language that can query RDF graphs. Anyone familiar with the SQL query language should find SPARQL easy to learn. SPARQL was designed to resemble SQL as much as possible. However, it is a very different kind of query language than SQL. Recall that RDF graphs are Subject ➤ Predicate ➤ Object triples. For example, `:Albert_Einstein :has_employer :Institute_For_Advanced_Study` is an example triple where `:Albert_Einstein` is the subject, `:has_employer` is the predicate, and `:Institute_For_Advanced_Study` is the object.

The power of SPARQL is that it can have a wildcard for any or all parts of a triple. Wildcards in SPARQL start with a ?. The SPARQL triple `?s ? p ?o` will match every triple in the graph, i.e., the entire knowledge graph. This is very powerful because it means systems can use SPARQL to discover information even when they don't know the schema of the model. For example, the triple `:Albert_Einstein ?p :Kurt_Godel` will return all the triples that link Einstein to Gödel. The triple `:Albert_Einstein ?p ?o` will return all the triples that have Einstein as a subject, i.e., everything that we know about Einstein in the graph database.

Labelled Property Graphs

A property graph is a type of knowledge graph, but not a Semantic Knowledge Graph. The difference between a property graph and an RDF knowledge graph is quite simple. In an RDF graph, the edges of the graph are implicit. There is no data structure in such graphs that actually stands for the edges. Rather the edges are inferred by property values. The triple `:Albert_Einstein :has_Employer :Institute_For_Advanced_Study` is implemented as a value on Einstein's `:has_Employer` property. That value is the individual `:Institute_For_Advanced_Study`. When a tool such as the AllegroGraph Gruff visualization tool used in Figure 1-1 draws the graph for an RDF triple, it draws each subject as a node in the graph, each property value as another node, and the property that connects them as an edge. This works the same way for OWL triples because OWL entities map very cleanly to RDF entities. An instance of an OWL class is the subject or object of a triple and OWL properties are triple predicates.

A property graph on the other hand has an explicit representation of each edge in the graph as well as the connected objects. The benefit of property graphs is that by explicitly representing the property, it is possible to have data on properties. For example, if we wanted to record the date that Einstein started work at the Institute and his title,

that could be modeled as additional properties on the `:has_employer` property (see Figure 2-3).

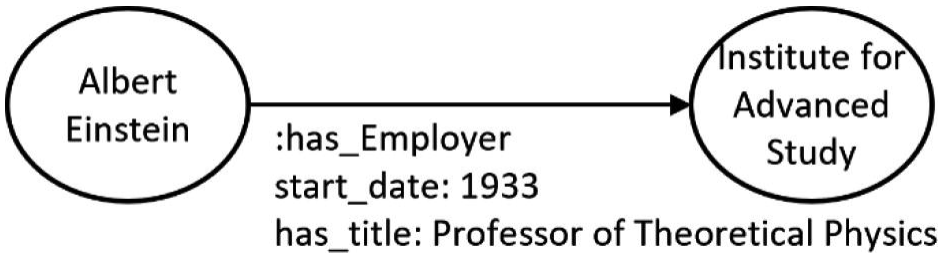


Figure 2-3 An Example Property Graph

A property graph differs from an RDF graph in how it represents links between nodes. In RDF, a link (or edge) is implied by the existence of a triple: the subject node is connected to the object node through a predicate. However, RDF does not treat the edge itself as a separate data structure that can have its own data. In contrast, property graphs explicitly represent each edge as an entity that can store additional information (such as a timestamp, weight, or label). It is possible to achieve the same modeling capability in OWL by utilizing the N-ary relation pattern[3] discussed in **Chapter 8**.

In addition to SHACL, the most important addition that was not part of the original Semantic Web Standards is RDF-star. RDF-star implements property graphs using quoted-triple support. RDF-star also requires extensions to SPARQL (called SPARQL-star) in order to take full advantage of the property graph formalism. While this has been incorporated into the RDF 1.2 standard suite by the W3C, parts of its formalization (e.g., entailment regimes, full SPARQL-star support) remain under development. RDF-star extends RDF with a native syntax for "quoted triples," which allows an RDF triple to appear directly as the subject or object of another triple. This provides built-in support for annotating statements. RDF-star simplifies many use cases where metadata needs to be attached to a triple, such as provenance, confidence scores, or temporal qualifiers.

Much of the demand for RDF-star was driven by the widespread success of non-RDF graph databases that support property graphs,

especially Neo4j. Neo4j's ecosystem, query language (Cypher), and market adoption make it the dominant reference point when discussing property graphs. A deeper exploration of property graphs and their role in modern semantic architectures is provided in later chapters.

Linked Data: A Distributed Web of Data

One of the earliest and most widespread successes of the Semantic Web vision is a concept known as Linked Data. Linked Data sites were created before the OWL standard was complete. For this reason and also for efficiency, since Linked Data graphs tend to be very large, Linked Data typically uses only RDF/RDFS and SPARQL rather than OWL. Linked Data allows web data to be connected, queried, and reused across domains, even when (in fact, this is the norm) each site was developed independently.

Linked Data is a global, distributed database. Instead of every website or application being an isolated silo, Linked Data promotes the idea that data on the web should be

- Identified using IRIs
- Described using shared vocabularies
- Interlinked with data from other sources
- Accessible via standard protocols (especially HTTP/HTTPS, RDF/RDFS, and SPARQL)

This approach enables powerful queries that can draw connections across several heterogeneous domains—from music to geography to scholarly research—without needing to centralize or preprocess the dataset. Linked Data sources like Wikidata, DBpedia, and GeoNames are often used to train or enrich large language models, because they provide large volumes of text, as well as machine-readable metadata about each text string.

Notable Linked Data Sources

Over the years, dozens of publicly accessible Linked Data sets have been created. Figure 2-4 shows a diagram of some of the most well-

known sites as well as common dependencies between them.

Here are a few of the most valuable and enduring:

- *Wikidata*: A structured, collaboratively maintained knowledge graph from the Wikimedia Foundation. Frequently used in both LLM training and semantic applications for data on just about any topic. https://query.wikidata.org/
- *GeoNames*: A massive geographic database with information on millions of places worldwide—linked to country, region, and population data. http://sws.geonames.org/
- *US Census Data*: Open United States government data on population, demographics, and economics—modeled as Linked Data. https://data.census.gov/
- *Project Gutenberg*: Linked data describing thousands of free public-domain literary works. https://www.gutenberg.org/
- *FOAF (Friend of a Friend)*: One of the earliest Linked Data vocabularies for describing people and social relationships. https://sparontologies.github.io/foaf/current/foaf.html
- *MusicBrainz*: An open music metadata database, especially valuable for linking artists, albums, and recordings. https://musicbrainz.org/
- *DBpedia*: DBpedia extracts structured data from Wikipedia infoboxes and has historically been a popular linked dataset. However, Wikidata is replacing it as the primary structured source derived from Wikipedia. https://www.dbpedia.org/

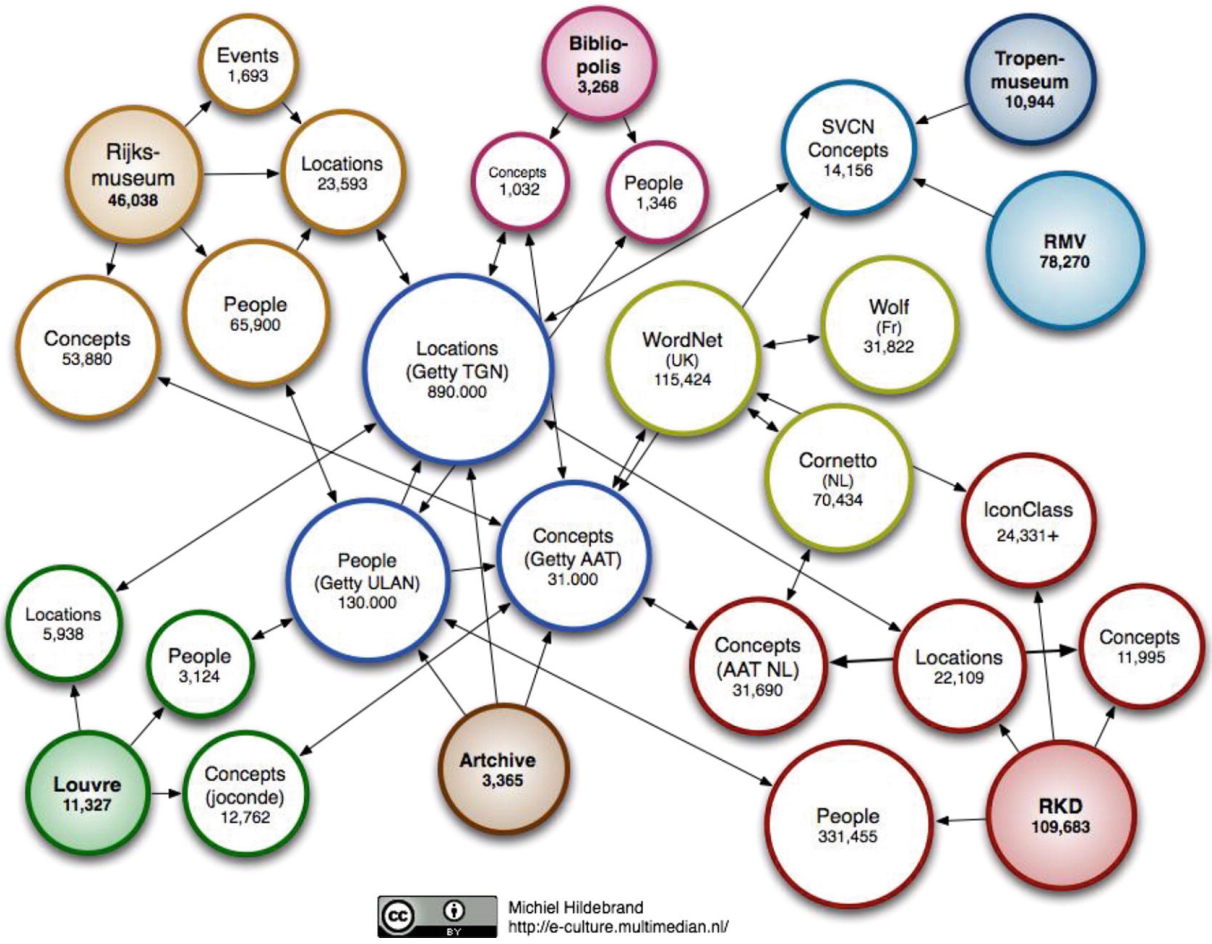


Figure 2-4 Graph of some Linked Data sites. Source: Wikimedia Commons

Figure 2-4 shows a Linked Data cloud diagram—offering a glimpse into the ecosystem of interlinked RDF datasets. The core idea is that data from different domains designed independently from each other can be linked, queried, and recombined using W3C standard technologies: HTTP/HTTPS, IRIs, RDF/RDFS, and the SPARQL query language.

Vocabularies

One of the greatest strengths of Semantic Knowledge Graphs is the ability to reuse existing vocabularies, which are carefully developed, widely adopted OWL ontologies that capture models developed by experts for common types of technical and business problems.

Rather than building every model from scratch, it is often more effective to incorporate these reusable vocabularies into your designs. Doing so brings two major benefits:

- *Efficiency*: Reusing proven models saves development time and avoids reinventing solutions to well-understood problems. It allows developers to focus on the problems and innovations that make their solutions unique while leveraging best practices to solve problems that are already well understood.
- *Interoperability*: Using widely recognized vocabularies makes it easier for your systems to integrate with other systems—and easier for others to understand, reuse, and extend your work.

We can broadly divide reusable vocabularies into two categories: technical vocabularies and domain vocabularies.

Technical vocabularies describe domains that cut across industries —areas where the same basic structures and concepts are relevant regardless of the specific business sector.

Examples include

- *Metadata*: Dublin Core (DCMI) (The Dublin Core Metadata Initiative (DCMI), 2024) and BIBO (Dublin Core Metadata Initiative (DCMI), 2024) for bibliographic metadata.
- *Provenance*: PROV-O (W3C Provenance Ontology) (Prov W3C Working Group, 2013) for tracking the history and derivation of data. Note: there are other versions of PROV-O that do not require OWL. However, we will always be using the ontology version, so from this point on, we will just call it PROV.
- *Knowledge Organization*: SKOS (Simple Knowledge Organization System) for managing taxonomies, thesauri, and classification schemes (W3C SKOS Working Group, 2005).
- *Security*: W3C Web Access Control (WAC) (World Wide Web Consortium (W3C), 2022) and other emerging models such as the Open Digital Rights Language (ODRL) (McRoberts & Doncel, 2014) for representing permissions and access policies.

Technical vocabularies are typically sanctioned and maintained by respected technical organizations such as the World Wide Web Consortium (W3C) or the Dublin Core Metadata Initiative (DCMI). Their broad adoption ensures stability and widespread community support. For example, Protégé and many other tools discussed below have built-in support for entities from common vocabularies such as SKOS and Dublin Core.

Domain vocabularies capture the semantics of specific business sectors—enabling precise modeling within fields like healthcare, finance, and commerce.

Examples include

- *Finance*: FIBO (Financial Industry Business Ontology), developed by the EDM Council (EDM Council, 2024)
- *Sales and Commerce*: GoodRelations (World Wide Web Consortium (W3C), 2015), a widely used ontology for product and service descriptions
- *Healthcare*: SNOMED CT (SNOMED International, 2024), a comprehensive healthcare ontology used worldwide to define medical codes known as clinical terminology

These domain vocabularies are usually created and maintained by industry consortiums or professional associations. Their models reflect the specialized needs of their respective fields and often align closely with regulatory requirements and best practices. Several such as SNOMED CT have histories going back decades, well before OWL. One of the benefits of OWL is that it provides a common language to model such vocabularies.

In the chapters ahead, we will explore how to select, adapt, and integrate reusable vocabularies into your Semantic Knowledge Graph designs—balancing reuse with the need for customization when modeling complex real-world systems.

Accessing Knowledge Graphs from Programming Languages

Applications that work with knowledge graphs typically access them from programming languages such as Python, Java, Lisp, or JavaScript. There are two approaches to doing this: using a graph database client or using a standalone programming library. Both approaches are valid; the appropriate choice depends on the requirements of the application.

Graph Database Clients

Most graph databases provide language-specific client libraries that handle communication with the database server. Examples include the Python and Java clients for AllegroGraph, the official Neo4j drivers, and the connectors for graph-enabled cloud platforms such as Amazon Neptune.

These clients expose the full functionality of the database, including

- Query execution (e.g., SPARQL, Cypher, PGQL)
- ACID transactions
- Concurrent access and locking
- Indexing and query optimization
- Authentication and security
- Server-side reasoning or constraint checking (when supported by the database)

When an application uses a full graph database, these clients are the natural and recommended choice. They provide the performance, reliability, and scalability needed for multiuser applications and enterprise systems.

Knowledge Graph Programming Libraries

In contrast, some applications do not require a full graph database. They may only need to load a knowledge graph into memory, manipulate it locally, and save it back to disk as a file. For these use cases, standalone graph libraries are often a better fit. Examples include

- *Python*: RDFLib (RDFLib Open Source Community, 2025), Owlready2 (LAMY, 2023)
- *Java*: Apache Jena Ontology API (Apache Software Foundation, 2025), Eclipse RDF4J (Eclipse Foundation, Inc., 2025)

These libraries implement the core RDF/RDFS and OWL data models directly in the programming language. They allow developers to

- Parse and write RDF and OWL files
- Manipulate triples and classes programmatically
- Run SPARQL queries locally (for libraries that support it)
- Perform basic reasoning (Jena, Owlready2)

Standalone libraries have several advantages:

- Very small footprint (no server to install or manage)
- Highly portable (works anywhere that Python/Java runs)
- Easy to integrate into local tools, scripts, or offline pipelines

However, they lack the full range of capabilities provided by a graph database—particularly concurrency, optimized indexing, large-scale storage, and transactional guarantees.

Both approaches are important: graph database clients are appropriate for applications requiring scalability, concurrency, and full database features. Standalone programming libraries are ideal for smaller tools, embedded applications, development environments, and lightweight workflows where a full server is unnecessary.

Because knowledge graphs are used in such a wide range of environments, from small scripts to enterprise systems, it is essential to recognize both access patterns. Later chapters explore these options in more detail, but the diagram in this chapter includes both to reflect the full landscape of real-world development.

Applications

One of the exciting things about the applications that can be developed with Semantic Knowledge Graphs is that in some ways, this section is

almost a blank slate. There are certainly many kinds of applications that have been developed with Semantic Knowledge Graphs, and we will provide examples of such systems in **Chapter 14**. However, since search and interaction with the graph are critical to all of these applications, the architectures used to build these in the past are now essentially obsolete, as LLMs will play a major role in such applications in the future and completely change for the better both the user interface (which can include natural language via the LLM) and the modeling of meaning, which can take advantage of the LLM capability to model meaning via vectors as well as the explicit knowledge representation of Semantic Knowledge Graphs. With that caveat in mind, here are some common applications for Semantic Knowledge Graphs:

- Semantics for Data Fabric architectures (Strengholt, 2023)
- Semantics for Data Mesh architectures (Dehghani, 2022)
- Semantics for data-driven architecture (McComb, 2019)
- Recommendation systems (Blumauer & Nagy, 2020)
- Semantic models for data lakes and data lakehouses (Dibowski et al., 2020)
- Semantic models for data virtualization (DeBellis et al., 2023)
- Data catalogs (Olesen-Bagneux, 2023)
- Retrieval-augmented generation (RAG) systems to utilize deep, curated domain knowledge with large language models (DeBellis et al., 2024) (DeBellis et al., 2025)

Summary

This chapter provided an overview of the technology utilized by Semantic Knowledge Graphs, most of which came from the Semantic Web initiative.

- The Semantic Web was launched by a paper by Tim Berners-Lee in Scientific American. He described how the original vision for the Internet had always been a Semantic Web; however, the technology

of the time couldn't support that vision, so a web of hypertext links was chosen as a compromise.
- Just as the United States Defense Advanced Research Projects Agency (DARPA) funded the Internet, so research funded by DARPA on knowledge representation was the catalyst for the Semantic Web.
- As often happens in research, the ultimate use of Semantic Web technology evolved from the original vision. The original vision focused on providing a semantic foundation for the Internet. While this has been achieved, the technology is at least as useful to organize enterprise data as to provide semantics for the Internet.
- The core technologies: Internationalized Resource Identifiers (IRIs), Resource Description Framework (RDF), Resource Description Framework Schema (RDFS), the SPARQL query language, the Web Ontology Language (OWL), and the Shapes Constraint Language (SHACL) are all standards defined by the W3C.
- The Protégé ontology editor is a free, open-source tool from Stanford University that is one of the most useful tools to design ontologies. It has plugins for SPARQL and SHACL that are useful as pedagogical tools; however, both plugins do not scale to the requirements of industry scale projects. For serious work, an implementation that supports the complete SPARQL specification and that can utilize and efficiently reason over large SHACL models is required. These can be found in the clients for graph database tools or programming libraries.
- Linked Data was the first big success of the Semantic Web. It provides a global collection of SPARQL end points that are an incredible asset, a globally distributed database that can be queried by anyone from anywhere using SPARQL. Linked data demonstrates one of the strengths of SPARQL: it can discover data and merge data from heterogeneous data sources with different data models.
- Labelled Property Graphs (LPGs) were not part of the original Semantic Web vision. However, in practice, they are a very powerful type of knowledge graph for certain applications. RDF-star is an emerging standard to provide property graph features as an

approved W3C standard. However, unlike all the other W3C specifications discussed here, RDF-star is not yet finalized. Neo4j is currently the dominant player in the space of Labelled Property Graphs, and their implementation is not based on RDF or IRIs.

- In addition to Linked Data, another success story of the Semantic Web is the creation of standardized vocabularies. A vocabulary is an ontology designed to be reusable that solves a common problem. These vocabularies are broadly divided into two categories: technical vocabularies and domain vocabularies. The best technical vocabularies are developed under the sanction of organizations such as the W3C and the Dublin Core Metadata Initiative. These include the Simple Knowledge Organization System (SKOS) for defining taxonomies and glossaries, PROV-O for data provenance, and Dublin Core for document metadata. The best domain vocabularies are created under the auspices of industry consortia such as SNOMED CT for healthcare medical coding, GoodRelations for eCommerce, and FIBO for financial services.
- There are two common ways to access a knowledge graph from a programming language such as Python or Java: using a general library or using the client provided by a graph database. Libraries are best for lightweight applications that need to load RDF files, run SPARQL queries in memory, or perform simple reasoning without requiring a full database. Graph database clients, by contrast, are designed for multiuser, scalable applications that benefit from persistent storage, ACID transactions, and server-side query processing. Most RDF-compliant graph databases provide a complete implementation of SPARQL and include strong SHACL support for data validation. Programming libraries typically provide full SPARQL support as well, but SHACL is less commonly built-in and often requires an additional tool.
- Semantic Knowledge Graphs (SKGs) are most commonly used in industry for analytic applications such as Data Catalogs, Data Lakes, and knowledge management. Another major trend is to use SKGs in

combination with large language models using architectures such as retrieval-augmented generation.

Footnotes

1 Examples include Linked Data, schema.org, and the GoodRelations ontology, all of which will be discussed in later chapters.

2 Decidable subset means it isn't as expressive as First-Order Logic (FOL), but, unlike FOL, you can write reasoners that will always reach a conclusion and terminate when reasoning over a valid model. In work that laid the mathematical foundation for modern computer science, Alan Turing and Alonzo Church proved that languages as expressive as FOL are undecidable: reasoners operating on them cannot be guaranteed to terminate. This is why early work in knowledge representation focused on decidable subsets of FOL. See DeBellis and Neches (2023) and DeBellis (2024) for details.

3 Property graphs do not support true n-ary relations in the formal logical sense. Every edge in a property graph is a binary relation between two nodes, and additional information such as timestamps, provenance, or roles is represented as attributes on that binary relation. However, from a practical data-modeling perspective, this mechanism gives users much of the expressiveness they would want from n-ary relations. Thus, although property graphs are not n-ary in the strict logical sense, they simulate n-ary relations in a convenient and intuitive way.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_3

3. Core Technologies

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

In this chapter, I describe the core Semantic Web technologies in detail and how they fit together and address different requirements. We will start by describing the graph level in more detail: IRIs, RDF, and RDFS. Then, we will discuss a new addition to the Semantic Web: Labelled Property Graphs. Then, we will describe the SPARQL query language, a powerful and essential tool for querying RDF graphs that glue all of the different components together.

Since all the layers of the Semantic Web are defined in RDF or languages built on top of RDF, SPARQL can query them all. Even more importantly, SPARQL can make changes to graphs and can transform one type of graph into another. For example, SWRL rules are implemented as RDF graphs and can be automatically transformed into SPARQL which is less intuitive but more efficient and widely supported than SWRL.

After explaining the basic SPARQL concepts, we will discuss and show some examples of Linked Data and how it can be queried with SPARQL.

Finally, we will conclude with the core concepts and examples for OWL, SWRL, and SHACL.

RDF/RDFS Semantic Networks

I'll begin by going into detail on the technologies that form the foundation for all other technologies: Internationalized Resource Identifiers (IRIs) are unique identifiers also called resources. The Resource Description Framework (RDF) is a language to describe graphs defined by IRIs and links to and from them. Resource Description Framework Schema (RDFS) is a language defined in RDF to describe datatypes, classes, and properties.

Internationalized Resource Identifier (IRI)

IRI stands for Internationalized Resource Identifier (IRI). An IRI is a unique identifier. Just as a URL (which, remember, is a kind of IRI) must be unique in order for a browser to take you to the correct place, so must any IRI be unique. An IRI points to the one and only one resource that it represents. The inverse is not true, however. Even for URLs, it can be the case that different URLs map to the same page. For example, it is often the case that https://website.com and https://website.com/index link to the same page. In the same way, two or more IRIs can refer to the same resource, and there are features in OWL to indicate that two classes or individuals are actually the same resource.

Anything with an IRI is called a resource. The Resource Description Framework (RDF) is a framework for describing resources on the Internet (and local Intranets). RDF describes resources as triples of IRIs. An RDF triple has a subject, predicate, and object. Each triple can be thought of as a very simple sentence where the subject and object are nouns (and nodes in a graph), and the predicate is a verb (an edge in the graph). In order to identify the specific subject, predicate, and object, we use IRIs. The IRI for the ontology is similar to the domain name in a URL. It provides the string that all entities in the ontology share. An example class in the People Ontology is the Person class with the IRI https://michaeldebellis.com/people/Person, and an example property is https://michaeldebellis.com/people/has_parent. Since the subject of one triple can be the object of another (and vice versa), a collection of triples defines a network graph.

The Resource Description Framework Schema (RDFS) is a vocabulary (a reusable model) that uses RDF to define a basic meta-model for properties, classes, collections, etc. Whereas RDF is a graph language, RDFS

is a Semantic Network language. Semantic Networks were one of the first forms of knowledge representation used in artificial intelligence (AI) (DeBellis & Neches, 2023). A semantic network is a network graph with typed nodes and links that have specific semantics (meaning). It is common to see these layers grouped together and referred to as RDF/RDFS. This is because languages such as OWL and SHACL use RDF/RDFS to define their specific vocabularies. Also, when a higher-level language such as OWL or SHACL isn't utilized as in Linked Data, RDF/RDFS are virtually always used together. Since RDF and RDFS are virtually always used together, from this point forward, for brevity, unless I need to distinguish between the two, I will just use the term RDF, and it should be understood to mean RDF/RDFS. Figure 3-1 shows a simple example of an RDF graph that models a few classes and individuals in the People Ontology. While the basic idea of an IRI is straightforward, there are important conventions and edge cases to be aware of — subtle details that, if overlooked, can lead to integration or interoperability problems.

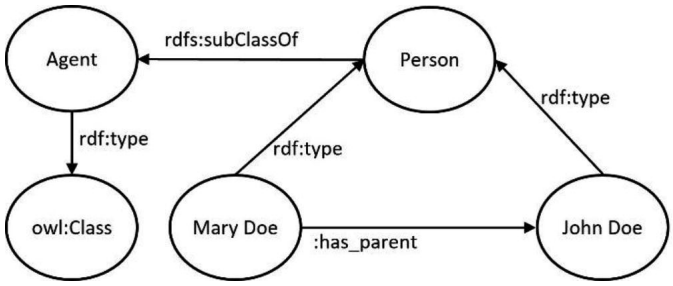


Figure 3-1 A simple RDF graph

In theory, IRIs support a wide range of Unicode characters, including non-ASCII letters and symbols. Unlike URIs, IRIs allow characters from different languages, such as

http://example.com/économie

http://例子.测试/资源

However, in practice, using non-ASCII characters, spaces, and special symbols in IRIs often leads to compatibility issues between tools. For example, an IRI that works in Protégé may be encoded differently in a graph database in order to maximize efficiency, causing unexpected results when importing/exporting between two different tools.

It is possible to use spaces and other special characters in IRIs using percent encoding. You've probably seen this in URLs. For example, if we wanted an IRI in the People Ontology that ended in "Jay Gatsby," we could use https://michaeldebellis.com/people/Jay%20Gatsby. However, using percent-encoded characters very often leads to compatibility and other issues. That is why for the People Ontology, I use underscores to separate words and names. Another standard that is very common is to just smoosh the words together using CamelCase. With CamelCase, we just capitalize each new word. This is nice because it cuts down on typing a bit. For example: https://michaeldebellis.com/people/JayGatsby. For readability, I recommend using underscores. One nice feature of Protégé is that when using user-defined IRIs, the default is if you type the name of an IRI with a blank, it automatically replaces it with an underscore. When we show some examples of how SPARQL can automate much of the work for designing and maintaining an ontology, we will see another reason that using underscores makes life a bit easier. However, CamelCase is a perfectly fine option as well. What matters most is not whether you use CamelCase or underscores but to pick one standard and use that consistently.

Another issue regarding the best practices to define IRIs is the question of which character to use to delimit the base IRI of the ontology from entity names. This character is called the fragment identifier. In the People Ontology, we use the "/" character as the fragment identifier. Until recently, the standard was to use the "#" character. Virtually all of the vocabularies that have been around for a while, such as the Simple Knowledge Organization System (SKOS), use the pound sign. We will discuss the SKOS property `skos:prefLabel` below. The full IRI for that property is `http://`www.w3.org/2004/02/skos/core#prefLabel. The reason people are adopting the slash delimiter is it is slightly more efficient for large

ontologies. When using the slash identifier HTTP can get individual entities. The pound fragment identifier requires the entire graph to be read at once, which can be slow for large graphs. The pound fragment identifier is slightly better for small graphs because it results in fewer HTTP requests. However, for small graphs, it doesn't really matter; you are going to have fine performance either way, and often, what starts out small ends up being big. I recommend you stick with the slash identifier all the time, which is the standard most people use now. You can configure this in Protégé using **File➤Preferences➤New entities** as shown in Figure 3-2.

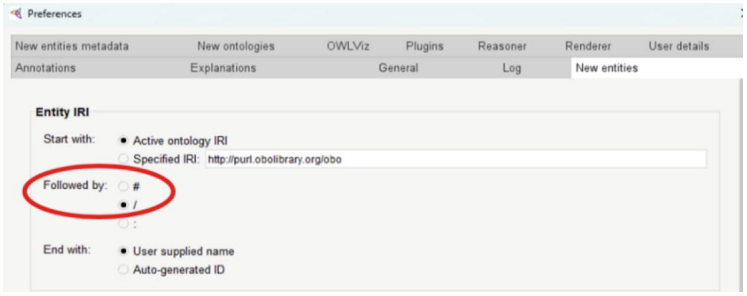


Figure 3-2 Configuring the Protégé preferences for new entities

Another issue to watch out for is whether your IRI starts with *http* or *https*. You might assume this doesn't matter because the difference concerns how data is transmitted, not the content itself. Unfortunately, RDF identifies resources purely by their character string, not by any network equivalence. Thus, http://michaeldebellis.com/people is a completely different IRI from https://michaeldebellis.com/people. Early Semantic Web vocabularies such as RDFS, OWL, and SKOS were created before *https* was widely adopted, so their official namespace IRIs begin with *http*. For any new IRIs you create, however, you should always use *https*, since it supports secure resolution and avoids later compatibility problems. You can also set this in Protégé using **File➤Preferences➤New ontologies**. That will provide a field where you can enter the default ontology IRI base. For example, in my environment, that is set to https://www.michaeldebellis.com/. Setting this parameter ensures that all new ontologies will consistently use HTTPS.

Prefixes

You may be wondering why the nodes and links in Figure 3-1 have short letters and colons before the name, such as `:has_parent` and `rdfs:subClassOf`. These are examples of prefixes. We use prefixes to make it easier to use IRIs in code and queries. If you've ever used namespaces or packages in programming languages such as Java, Ada, or Lisp, you are already familiar with this concept. An ontology file or query where each entity has an IRI such as `http://`www.w3.org/2000/01/rdf-schema#subClassOf would be cumbersome to deal with. Prefixes are defined at the start of the file or query, and from that point on, any reference to an IRI in that namespace can be done with a prefix.

To see examples of prefixes, we'll look at a simple excerpt of what Protégé generates to save an ontology. There are different formats for saving OWL files. These are called streaming formats. The streaming format that used to be most popular was RDF/XML. However, like XML in general, it is going out of favor because it is too verbose. The most popular formats now are Turtle and JSON-LD (although all tools still support RDF/XML since many older vocabularies use that format). We'll use Turtle because in addition to being less verbose than RDF/XML, it is very similar to the syntax of the SPARQL query language. If you open up the People Ontology in a simple text editor such as Notepad++, at the top of the file, you will see declarations as shown in Listing 3-1.

```
@prefix owl:  <http://www.w3.org/2002/07/owl#>.
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <https://michaeldebellis.com/people/>.
```

Listing 3-1 Examples of Prefix Declarations

Prefixes allow you to reference an IRI with just the last part. For example, to use the RDFS IRI to refer to the `subClassOf` predicate `http://`www.w3.org/2000/01/rdf-schema#subClassOf, you can simply write `rdfs:subClassOf`.

Note that the ontology itself has no string (i.e., the empty string) for its prefix. This is a common convention so that one can refer to the entities in the ontology with just a colon with no string in front of it. For example, with the prefixes above, we can define classes such as Person and Agent as `:Person` and `:Agent`. The simple model described in Figure 3-1 could be described (using these prefix definitions) in Turtle as shown in Listing 3-2.

```
:Agent rdf:type owl:Class;
       rdfs:subClassOf owl:Thing.
:Person rdf:type owl:Class;
        rdfs:subClassOf :Agent.
:has_parent rdf:type owl:ObjectProperty;
            rdfs:domain :Person ;
            rdfs:range :Person.
:Mary_Doe rdf:type :Person;
          :has_parent :John_Doe.
```

Listing 3-2 Turtle Code for the Model in Figure 3-1

In Turtle (and SPARQL queries), a semi-colon at the end of a triple means the next triple below it has the same subject as the previous row. This is what developers call "syntactic sugar," a convention that makes a common situation easier to define. We often define (or want to match) many properties for the same subject at the same time. A period means the end of definitions for a specific subject. Thus, the next line after a period always requires a new subject, even if it is the same as the line before. The first two lines in the above definition are equivalent to those shown in Listing 3-3.

```
:Agent rdf:type owl:Class.
:Agent rdfs:subClassOf owl:Thing.
```

Listing 3-3 Turtle Code for Agent Without Using Semi-colon "Syntactic Sugar"

Without the prefixes, the two triples above would be as shown in Listing 3-4:

```
<https://michaeldebellis.com/people/Agent> <http://www.w3.org/1999/02/22-
rdf-syntax-ns#type> <http://www.w3.org/2002/07/owl#Class>.

<https://michaeldebellis.com/people/Agent>
<http://www.w3.org/2000/01/rdf-schema#subClassOf>
<http://www.w3.org/2002/07/owl#Thing>.
```

Listing 3-4 Without Prefixes, Queries and Turtle Code are Cumbersome

In Turtle and SPARQL, each full IRI is enclosed in angle brackets, while prefixed names such as `rdf:type` are written without them. The brackets are part of the syntax, not part of the IRI itself. As you can see, without prefixes, Turtle and SPARQL would be very cumbersome to read and edit.

IRIs and Labels

A label is just a string property that is the "pretty name" for an entity. Unlike an IRI, a label does not have to be unique in either direction. A resource can have more than one label, and different resources may have the same label. Yet again, the basic concept is simple, but the devil is in the details.

One issue that is somewhat controversial is whether or not to use a user-supplied name or an auto-generated ID for your IRIs. Look back at Figure 3-2, and you will see below where you define whether to use a `#` or `/` there is an option called **End with**. It is "end with" because the first part of the IRI will always be the

IRI for the ontology itself followed by a # or /. The parameter you select in **End with** determines whether your IRIs will be automatically generated or defined by the user.

When developers use auto-generated IRIs, they have two options. These will show up if you click **Auto-generated ID** in the **New Entity Preferences**. Those two options are **Numeric (Iterative)** and **Globally Unique**. With numeric IRIs, you start with an alphanumeric code such as `OBO_` and you append a number to that code every time you create a new IRI. You also specify how many digits to use for the number. The default is seven. If we used that string with seven digits, and the name of our ontology was `http://ontology_domain.com/example`, the first entity we created would have the IRI `http://ontology_domain.com/example/OBO_0000001`. If you select **Globally Unique**, then Protégé will use an algorithm to generate a Universally Unique Identifier (UUID). This is a long alphanumeric string that is guaranteed to be unique. Protégé will append a UUID to the string `OWL_Class` or `OWL_Property`, etc., depending on the kind of entity you are creating. For example, if you create a class, the string will look something like `http://ontology_domain.com/example/OWLClass_f861e81c_661a_4243_a9be_cb9c780cb78a`. You won't see that of course. When you use auto-generated IDs, you type a string with an intuitive name, and that string is set as the label for the new entity. The problem is that when you work at the level of code, SPARQL, and SHACL, you need to work at the IRI level, not the label level. As a result, if you use auto-generated IRIs, you end up with SPARQL queries and code in languages such as SHACL and Python that look like the query shown in Listing 3-5.

```
SELECT ?p ?t
WHERE {?p rdf:type
:OWLClass_f861e81c_661a_4243_a9be_cb9c780cb78a; :OWLProperty_c744v9fv_594j_364
?t.}
```

Listing 3-5 SPARQL Query with UUIDs (Why Programmers Prefer User-Defined IRIs)

This violates one of the most important principles of good programming: self-documenting code. With user-defined IRIs, this query becomes that shown in Listing 3-6.

```
SELECT ?p ?t
WHERE {?p rdf:type :Patient;:has_temperature ?t.}
```

Listing 3-6 Same Query with User-Defined IRIs

Especially with modern IDEs that can auto-complete names as you type them (and with LLM support even *before* you start to type them!), it is a best practice to use long names that clearly describe what a variable is rather than a short name with a comment that explains it.

The Semantic Web community is divided between those who believe in auto-generated IRIs and those who believe in user-defined IRIs. The divide aligns very well with industry vs. academic and ontologist vs. ontologist/developer. People in industry usually use user-defined names. People who design reusable ontologies and don't write code tend to prefer auto-generated names. People who do both, like me, virtually always use user-defined names. I'm not going to go into the issue further here. If you are interested, see DeBellis (2020) for more details. I strongly encourage you to always use intuitive, user-defined IRIs, and that is what we will use for all examples we build related to this book. There are three exceptions where auto-generated opaque IRIs are justified:

1. *Individuals*: Typically, we don't manually define individuals; we generate them from data such as exports from relational databases. Also, we typically don't reference specific individuals in SHACL, SPARQL, or code in languages like Python and Java that access our models. Thus, when defining individuals, it is almost universal across both industry and academia to use auto-generated IRIs, typically UUIDs. Note that for the People Ontology, I didn't do this because that ontology doesn't have much data and the data was defined manually because it is meant for training not actual use.
   Standards: There are large organizations that support curated ontologies and that have adopted codes

2. standards. There are large organizations that support curated ontologies and that have adopted codes or other opaque formats for IRIs. Two examples are Wikidata and the OBO Foundry. In such environments, it is often mandatory to use a certain type of IRI. Even when it isn't mandatory, conforming to standards is still one of the most important success factors for a project, and thus, generated IRIs are encouraged in these environments.
3. *Security*: For some types of systems, having opaque IRIs can add another layer of security.

Note that while I recommend using user-defined IRIs for the ontology, these IRIs are not meant to be "pretty names" that users can understand but rather identifiers that are intuitive to developers. The names with appropriate spaces, special characters, and multilingual support are the labels which we discuss next.

Defining Labels

Labels are defined using the RDFS property: `rdfs:label`. As we will discuss in the next section, it is possible to have two different OWL individuals or classes that are really the same logical entity. However, this should be reserved for special cases. For most use cases, if you have more than one name for an entity, you should just use different labels. The Simple Knowledge Organization System (SKOS) vocabulary provides additional properties that are perfect and virtually always used for this: `skos:altLabel`, `skos:prefLabel`, and `skos:hiddenLabel`. You use `skos:altLabel` for synonyms that are not the preferred name but that you want to record as valid names for the entity. You typically then reserve `skos:prefLabel` for the label that is the preferred label when displaying the entity. Although it isn't used as much, `skos:hiddenLabel` is useful to record other names for an entity that may occur in systems or documentation but that you never want to use when displaying the entity. Note: these are standards, and most tools give you the freedom to do as you wish with your code and configuration of the tools. However, it is best to conform to these standards to make your work more reusable and able to take advantage of tool capabilities without additional configuration. In Protégé, you can add label properties and define the order for them using the option **File ➤ Preferences ➤ Renderer ➤ Configure**. Figure 3-3 shows this for the People Ontology. We have three label annotations: `skos:prefLabel` is the preferred label, then `rdfs:label`, and finally `skos:altLabel`.



Figure 3-3 Defining label preferences in Protégé

Resources May Have More Than One IRI

In RDF and OWL, entities are not assumed to be unique. Because OWL is meant to model large data sets that often evolve independently, it has a more flexible model than most databases. This means that, if we find two IRIs that actually model the same individual, we have the means to logically define that. This is why OWL has two properties: `owl:sameAs` to indicate that two individuals with different IRIs are really the same and `owl:differentFrom` to explicitly tell the reasoner that two individuals are different. An example of the first from the People Ontology is the pair of individuals `Jay_Gatsby` and `James_Gatz`; see Figures 3-4 and 3-5.

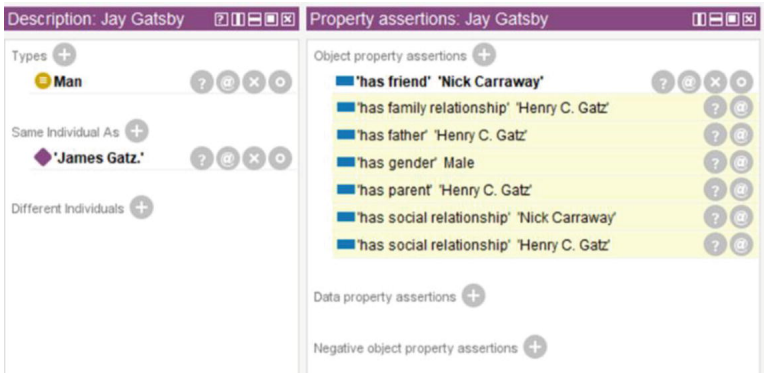


Figure 3-4 Property values for Jay Gatsby

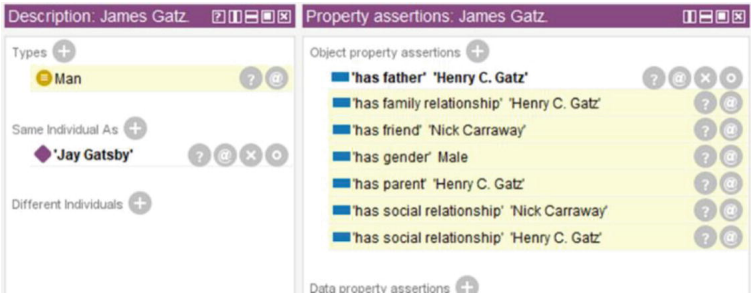


Figure 3-5 Property values for James Gatz

In the book *The Great Gatsby*, Gatsby's real name was James Gatz—but in some ways, they were different people. Gatz had very humble, unsophisticated roots that Gatsby was ashamed of and tried to hide. You can see the subtle difference in the two definitions. Even though they are logically equivalent, the direct user assertion that Gatz's father was Henry C. Gatz is made on the James Gatz individual because this was something Gatsby was ashamed of and tried to hide, although people from his humble origins who knew him as Gatz knew who his father was. On the other hand, the assertion that Nick Carraway was a friend of Gatsby is made on the Gatsby individual because Carraway didn't know his friend's original name was Gatz until the very end of the novel. However, once they are declared to be the same individual, the reasoner has —for both individuals—integrated the logical axioms asserted on each over to the other (the assertions highlighted in yellow). The same can be done with classes using the **Equivalent To** option discussed in the section below on classes.

The `owl:differentFrom` property does the opposite; it tells the reasoner that two individuals are different. In some cases (e.g., some SWRL rules and OWL axioms about cardinality of properties), it is necessary to know that two individuals are different in order to get the proper inferences. Protégé includes a very useful command: **Edit➤Make all individuals different**. This will make every individual in the ontology at that point different from each other.

Labelled Property Graphs (LPGs)

Labelled Property Graphs (LPGs)—often simply called property graphs—have gained significant traction in recent years as a way to define and implement knowledge graphs. Like other knowledge graphs, property graphs allow for expressive, dynamic relationships between entities. This flexibility has made them a powerful tool in applications that leverage graph-theoretic algorithms where the explicit modeling of an edge can make an algorithm much more intuitive to implement.

The "Labelled" in Labelled Property Graph refers to the fact that nodes and edges (also called vertices and relationships) are annotated with labels on nodes. These labels are essentially type tags—e.g., a node might have the label Person, or multiple labels like person and employee. These are not keys or values; they

are categories. In addition, labels are used to define data types on edges (sometimes called "relationship types"). Edges have a single type label (e.g., `has_employer`) that defines the semantic nature of the connection—just like a predicate in RDF. Unlike RDF, however, property graphs allow data to be attached directly to the edge itself. This means you can store additional information such as `start_date` or `has_title` as attributes of the relationship without needing to reify it as a node, as you would in RDF (or as a class as you would in OWL). Reification is a fancy term that just means creating a class that stands for a relation. It is a design pattern that most OOP developers are familiar with and can also be used with OWL. We'll discuss it in the section on ontology patterns in **Chapter** 8.

Unlike RDF, which is backed by formal W3C standards, property graphs currently lack a centralized standards body. However, Neo4j has emerged as the de facto standard in this space due to several key factors:

- *Market Dominance and Industry Adoption*: Neo4j is by far the most widely used property graph database, with broad uptake across enterprises, startups, and research labs. Many organizations that work with property graphs default to Neo4j.
- *Cypher Query Language*: Cypher, Neo4j's declarative graph query language, has become the most recognized and widely used query language for property graphs. The openCypher project, initiated by Neo4j, has influenced other graph systems, including RedisGraph, Memgraph, and SAP HANA Graph. Meanwhile, the ISO working group on graph query languages is developing Graph Query Language (GQL), which incorporates Cypher-like syntax and is expected to become an official standard.
- *Robust Ecosystem and Integrations*: Neo4j provides strong developer support, including libraries and drivers for Python, Java, and other languages, as well as cloud-native deployment options. Major cloud platforms—AWS, Google Cloud, and Azure—offer native or managed support for Neo4j. Additionally, the Neo4j Labs Neosemantics (n10s) plugin enables limited RDF interoperability, allowing organizations to integrate Semantic Web technologies with the property graph model.

RDF-star and Property Graphs

RDF-star (previously called RDF*) has emerged as an extension of RDF that introduces Labelled Property Graph (LPG) capabilities into Semantic Knowledge Graphs. RDF-star allows users to add data to properties, making it functionally similar to property graphs but within the RDF framework. Although RDF-star is not yet a finalized W3C standard, several vendors have already adopted it, including

- *TopQuadrant (TopBraid EDG)*: Provides RDF-star support for modeling statements about statements, with tooling that enables fine-grained metadata, data catalog annotations, and integration with SHACL-based workflows.
- *AllegroGraph*: One of the first triple stores to provide RDF-star support, allowing data on relationships while maintaining RDF/OWL compatibility. In fact, AllegroGraph supported this functionality even before the RDF-star standard.
- *GraphDB*: Implements RDF-star to extend the expressiveness of RDF-based data models.
- *Blazegraph*: Supports RDF-star to enhance RDF querying and representation.
- *Apache Jena*: Offers full RDF-star support in its core framework and Fuseki server, including quoted triples in Turtle-star and TriG-star syntaxes, and end-to-end API support for embedded triple modeling and querying.

These vendors offer property graph functionality within an RDF environment, allowing users to take advantage of standards compliance and interoperability while benefiting from more flexible relationship modeling. However, since OWL reasoners are designed for binary relations, they do not natively support reasoning over RDF-star annotations. This means that while RDF-star allows richer relationship modeling, it does not integrate fully with OWL's inferencing capabilities. Instead, RDF-star can be used alongside OWL, with OWL governing logical inferencing while RDF-star provides additional expressivity at the graph level.

In order to fully exploit the capabilities of RDF-star, extensions to SPARQL are required. This new version of SPARQL is called SPARQL-star. It extends the SPARQL query language to support embedded triples as first-class citizens, allowing users to query and manipulate metadata about triples directly. For example,

SPARQL-star enables you to write patterns where a triple is used as the subject of another triple, simplifying queries that involve provenance, annotations, or other contextual information. While traditional SPARQL required reification to express such constructs, SPARQL-star provides a more concise and intuitive syntax. Both RDF-star and SPARQL-star are currently (2026) Working Drafts under active development by the W3C RDF-star Working Group, but they are already supported by several RDF stores and are gaining traction in real-world applications.

While SPARQL-star introduces a way to query embedded triples in RDF-star, its functionality is conceptually similar to that of Cypher, the query language used in Neo4j. Cypher allows data to be attached directly to relationships, which is functionally equivalent to annotating RDF triples in RDF-star. However, SPARQL-star maintains compatibility with RDF's semantics and the broader W3C ecosystem, while Cypher is not part of any W3C standard and is based on a different graph model. Each language reflects the design priorities of its ecosystem: SPARQL-star emphasizes semantic richness and standardization, whereas Cypher emphasizes developer convenience and performance in graph operations. SPARQL-star is also not yet an approved standard but as with RDF-star, some vendors already have implementations:

- *Apache Jena (Fuseki)*: One of the most complete implementations of SPARQL-star, including support for embedded triple patterns, SPARQL-star UPDATE, and all *star* syntaxes (Turtle-star, TriG-star).
- *AllegroGraph*: Provides full SPARQL-star support, allowing embedded triple queries, updates, and reasoning over quoted triples. AllegroGraph implemented these capabilities early in the evolution of the standard.
- *GraphDB*: Implements SPARQL-star querying and updates, with native support for embedded triples and visualization of triple-term structures in its user interface.
- *Blazegraph*: Offers partial SPARQL-star support, including querying over embedded triples, although development of the platform itself is no longer active.

Other Labelled Property Graph Databases

Though Neo4j dominates, other Labelled Property Graph (LPG) databases exist, each with varying levels of adoption:

- *TigerGraph*: A high-performance, proprietary property graph focused on large-scale analytics.
- *Amazon Neptune*: A cloud-based graph database supporting both RDF and LPG models, though its LPG adoption lags behind its RDF capabilities. Neptune's LPG model is an open-source model from Apache called TinkerPop. TinkerPop uses an (also open source) query language called Gremlin.
- *JanusGraph*: An open-source, distributed property graph database, but with a smaller market presence.
- *ArangoDB & Memgraph*: Alternative graph solutions with property graph features but less adoption than Neo4j.

Choosing Between Semantic Knowledge Graphs and Property Graphs

If you're new to graph-based data modeling, it's natural to ask: when should I use a Semantic Knowledge Graph (SKG) based on OWL and RDF, and when is a Labelled Property Graph (LPG) the better choice? I spent much of my career as a consultant, and the most important two words of consulting speak I learned were "it depends." The answer depends on your requirements, your data, and your architectural preferences.

Semantic Knowledge Graphs follow W3C standards (RDF, RDFS, SPARQL, SHACL, SWRL, and OWL) and emphasize interoperability, semantic clarity, and extensibility. They are ideal when you're managing knowledge across domains, linking disparate datasets, or building a long-term enterprise data integration and reasoning strategy.

By contrast, property graphs are often more developer-friendly for certain applications, especially when performance, graph traversal, or large-scale data pipelines (e.g., in a Hadoop environment) are top priorities. Table 3-1 summarizes the features, strengths, and weaknesses of the two.

Table 3-1 Semantic vs. Property Knowledge Graphs

The Strengths of Semantic Knowledge Graphs

Semantic Knowledge Graphs are uniquely positioned for data-centric architectures that prioritize interoperability, longevity, and semantic clarity. OWL and RDF offer exactly the kind of self-describing, discoverable, and decentralized data architecture that Data Fabric, Data Mesh, and Data Lakehouse models aspire to. These benefits cannot be overstated. As organizations increasingly adopt these strategies, it's surprising and unfortunate that there is not more discussion of the value added by Semantic Knowledge Graph technology. This is likely due to lack of familiarity on the part of practitioners—a deficit this book hopes to address. The benefits of using IRIs to identify and relate data entities cannot be overstated.

Semantic Knowledge Graphs are also ideal for Data Catalogs, where the ability to link metadata, policies, standards, and provenance in a flexible yet precise way is essential. And for knowledge management, especially in scientific or regulated domains, the explicit semantics and inference capabilities of OWL provide both power and rigor.

When to Consider Property Graphs

The strengths of property graphs are for algorithmic graph problems like shortest path, community detection, or influence scoring. They integrate smoothly into Hadoop and Spark-based pipelines, and their traversal-centric query languages like Cypher often feel more natural to developers focused on navigating relationships rather than reasoning about them.

The Future of Property and Semantic Knowledge Graphs

Today, Semantic Knowledge Graphs and Property Graphs are divergent approaches—each with their own modeling style, query language, and developer community. But if we zoom out and consider the history of technology adoption, a familiar pattern emerges.

Whenever standards-based models compete with proprietary or pragmatic alternatives, we typically see early fragmentation, followed by gradual convergence as the ecosystem matures. SQL itself began as a standard that vendors extended with custom features, before the core became solidified and broadly adopted. We've seen similar convergence in web technologies, where once-competing approaches like JavaScript frameworks and HTML standards now interoperate more harmoniously.

It is highly likely that Semantic Knowledge Graphs and Property Graphs will follow a similar trajectory. As more organizations demand interoperability, semantic richness, and developer-friendly tooling, we can expect the eventual emergence of a unifying graph model or framework—one that combines the logical rigor of Semantic Knowledge Graphs with the agility and performance of Property Graphs.

Already, we're seeing early signs of this convergence. Some vendors now offer hybrid support (e.g., RDF + property graphs), and projects are underway to define standards that bridge the gap—for example, by extending RDF-star to better support property-style annotations. For now, the choice between these graph models depends on your specific goals. But long-term, the distinction may become more fluid as the best of both worlds comes together in a common foundation.

While Neo4j currently dominates, the landscape of property graphs continues to evolve. The development of GQL as an official ISO standard could introduce more competition and interoperability among property graph databases. Additionally, RDF-star adoption is growing; Semantic Knowledge Graph databases increasingly support property graph-like capabilities natively.

As a result, the gap between property graphs and Semantic Knowledge Graphs will most likely continue to narrow, leading to more hybrid approaches that integrate fast graph traversal with logical inferencing.

SPARQL: The Query Language for RDF Graphs

When working with a knowledge graph, one of the most important things is how it facilitates discovery—finding out what relationships exist between entities, even when we don't already know them. SPARQL is the tool that lets us do this.

SPARQL is a recursive acronym like GNU (GNU is Not Unix). It stands for SPARQL Protocol and RDF Query Language. SPARQL is the query language for RDF. Like SQL for relational databases, SPARQL helps us find and manipulate data. However, instead of working with tables, SPARQL operates on graphs defined by

triples—statements of the form Subject ➤ Predicate ➤ Object. This is just an overview of SPARQL. A good next step after this book to learn more about SPARQL is DuCharme (2013), and the W3C SPARQL specification provides the details of the entire query language as well as examples of how to use it (Harris & Seaborne, 2013).

Structure of a SPARQL Query

First, we will describe the basic structure of a SPARQL query, then we will illustrate these structural concepts with simple examples from the People Ontology and the Snap SPARQL Protégé plugin. However, a caveat before we start: Protégé is a tool to model OWL. Its support for SPARQL and SHACL is thus intended only for pedagogical purposes, not for real-world use. The SPARQL tools in Protégé do not implement the entire specification and leave out many critical capabilities such as INSERT and DELETE statements and path operators. When the time comes to explore more complex queries, I'll introduce you to tools which implement the complete SPARQL specification, such as the Wikidata Query Service (WDQS). If you are familiar with SQL, you will find that almost all of the major keywords in SPARQL are terms from SQL and provide almost identical features except for graph data models rather than table data models.

A basic SPARQL query has three main parts:

1. *Prefixes*: Shortcuts for IRIs to make queries more readable. Typically, we place all the prefixes at the top of the SPARQL query and leave them in place as we enter different queries.
2. *Operation Clause*: What to do with the results. `SELECT` displays the matching data, `INSERT` and `DELETE` add or delete matching data, and `CONSTRUCT` creates new triples but doesn't add them to the graph. The most common operation is `SELECT *` which will return all the triples that match any wildcards.
3. *WHERE Clause*: Defines what parts of the graph match based on triples with variables (wildcards that start with `?`).

In addition, the WHERE clause can have structure:

1. *Nesting*: It is possible to embed `WHERE` clauses within other `WHERE` clauses.
2. `OPTIONAL` *Clauses*: Normally, if there isn't at least one match for each wildcard, then the query fails and nothing is returned. Putting triples inside an `OPTIONAL` clause means that those values get bound if they can, but if they aren't bound, the query can still succeed. Among other things, this can be useful to clean up data and find graphs where data should exist but doesn't.
3. *Aggregation*: At the end of the `WHERE` clause, there are various keywords that can define ways to group the data.
4. *Solution Sequence and Modifiers*: At the end of a `WHERE` clause, there can be keywords such as `LIMIT` and `ORDER BY` that tell SPARQL to limit the number of results or to order them in a specific way.

For the following, make sure you have Protégé open, the People Ontology loaded, and the **Snap SPARQL** tab visible.

Basic SPARQL Queries

SPARQL queries work from the inside out. The first thing to run isn't the beginning of the query but rather the `WHERE` clause. Whenever you need to write or understand a SPARQL query, that's where you start. The queries used here can be found at https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Listings_3-7_to_3-13_Basic_SPARQL_examples.rq.

Delete the contents in the current Snap SPARQL tab and instead copy/paste the first example from the example file (Listing 3-7).

```
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX :  <https://michaeldebellis.com/people/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
WHERE {?class rdf:type owl:Class}
```

Listing 3-7 SPARQL Query to Retrieve All Classes

Starting from the WHERE clause, the subject of this triple is a wildcard: `?class`. The predicate is `rdf:type` and the object is `owl:Class`. This simple query will match everything that has the type class. That is, all the classes in the ontology.

That query lists the classes by their IRI. Another way to find classes is using the `rdfs:subClassOf` property. Next, let's use that and also view their label and put them in alphabetical order based on the labels. As with all SPARQL queries, I'm not going to list the prefixes again unless we need new ones but remember they always need to be there. That is, for this and all following queries, delete the previous query starting at `SELECT` but leave the four `PREFIX` lines.

Recall that when you end a line in SPARQL or Turtle with a semi-colon, that means that the subject for the next triple is the same as the previous triple. Thus, the query in Listing 3-8 binds `?class` to all subclasses of `owl:Thing` in the first line, then the second line finds the `rdfs:label` value for each IRI bound to `?class`. That is, the second line also has `?class` as the subject.

```
SELECT *
WHERE {?class rdfs:subClassOf owl:Thing;
              rdfs:label ?class_label.} ORDER BY ?class_label
```

Listing 3-8 SPARQL Query to Retrieve Classes and Labels

We do a similar query to find all the object properties and their labels (Listing 3-9).

```
SELECT ?label
WHERE {?object_property rdf:type owl:ObjectProperty;
                        rdfs:label ?label} ORDER BY ?label
```

Listing 3-9 SPARQL Query to Retrieve Object Properties and Labels

Let's do a new query where we get instances this time. Let's find all the instances of `:Person` that have a value for the `:has_age` property. Also, we'll modify the `SELECT` statement so that rather than returning all the wildcards, it only includes the `?person_label` and `?age`. Paste the query shown in Listing 3-10 into Snap SPARQL.

```
SELECT ?person_label ?age
WHERE {?person rdf:type :Person;
               rdfs:label ?person_label;
               :has_age ?age.} ORDER BY ?age
```

Listing 3-10 SPARQL Query to Retrieve People Who Have Age Values Defined

This only returns a few people instances because we have only defined a few values for the `:has_age` property. If we want to include ***all*** the people and show their age if it is available, that's where we need `OPTIONAL`. Delete the previous query and paste the one shown in Listing 3-11.

```
SELECT ?person_label ?age
WHERE {?person rdf:type :Person;
               rdfs:label ?person_label.
OPTIONAL {?person :has_age ?age}} ORDER BY ?person_label
```

Listing 3-11 SPARQL Query to Retrieve All People with Ages When Available

The `OPTIONAL` clause is a way to find data that may not always be there. Without the `OPTIONAL` clause, all the wildcard variables such as `?label` and `?age` ***must*** bind to some value for the query to succeed. So, if we want to find all the people, as well as the ages for the few that have them, we need to put the triple pattern to match for age inside an `OPTIONAL` clause. That way, for each individual that binds to `?person`, a value for their age data property will show up if present, but we will still see all the people either way.

Paste the query shown in Listing 3-12 to see everyone with a birthday date.

```
SELECT *
WHERE {?person :birth_date ?birth_day}
```

Listing 3-12 SPARQL Query to Retrieve People with Known Birthdays

Note that in this query, you didn't need to specify the `rdf:type` of `?person`. Because in the People Ontology, only instances of `Person` have an age, we can just match the triple and know that it will only match people. If you wanted to see all the people as well as the age and birth date for those that have them, you would use the query shown in Listing 3-13.

```
SELECT ?person_label ?age ?birth_day
WHERE {?person rdf:type :Person;
               rdfs:label ?person_label.
OPTIONAL {?person :has_age ?age}
OPTIONAL {?person :birth_date ?birth_day}} ORDER BY ?person_label
```

Listing 3-13 SPARQL Query to Retrieve All People and Birthdays and Age When Known

Note an important subtlety: we had to create ***two*** `OPTIONAL` statements. If we had included both the birth date and age in the ***same*** `OPTIONAL` statement, then the same rules for success of a query would hold within that `OPTIONAL` statement. We would only get the age and birthday for individuals who had them ***both*** defined.

SPARQL Queries to Generate Labels

While Snap SPARQL has many limits, it is still possible to do some useful things with it. This shows an example with queries that are very useful to generate labels from IRIs. When you set Protégé to user-generated IRIs, it doesn't fill in a string for the label. It can be a pain to constantly remember to enter a string every time after you create a new entity. Besides being extra work, entering the labels manually increases the chances of errors. For ontologies that follow the standard used in this book, where words are separated by underscores in the IRI, the following queries generate labels for all classes, properties, and individuals.

To see this example, load the following ontology into Protégé: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Pattern_Examples_No_Labels.ttl .

This is an example of an ontology I created in Protégé with user-defined IRIs and no labels. It implements some of the patterns in the book *Analysis Patterns* by Martin Fowler (Fowler, 1997). I developed it to show how well many object-oriented analysis patterns can be modeled in OWL. However, so it could serve as an example here, I've skipped my normal step of using SPARQL to generate labels for it. If you look at properties such as `:birth_Place`, `:is_contained_in`, `:first_Name`, `:last_Name`, as well as individuals such as `:Nelson_Mandela`, you will see they are all displayed with underscores because there are no labels defined.

We can automatically generate labels using the queries in the file: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Listing_3-14_SPARQL_Label_Generation_Updates_Patterns.ru .

First, remember to paste the `PREFIX` mappings at the top of the file in the top of the query box for Snap SPARQL and leave them there for all the queries. Then, start with the following query shown in Listing 3-14 to generate labels for classes.

```
# SPARQL Query to Generate Labels from IRIs
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX :  <https://michaeldebellis.com/pattern_examples/>

CONSTRUCT {?class rdfs:label ?lblname.}
WHERE {?class a owl:Class.
      BIND(STRAFTER(STR(?class), 'pattern_examples/') as ?name)
      BIND(REPLACE(?name,"_", " " ) as ?lblname)
      OPTIONAL{?class rdfs:label ?elbl.}
      FILTER(?class != owl:Thing && ?class != owl:Nothing && "" != ?lblname
&& !BOUND(?elbl))}
```

Listing 3-14 SPARQL Query to Generate Labels from IRIs

The very first line is a comment. As with Python, a pound sign in SPARQL is a comment that will be ignored by the SPARQL engine. After the comment, we want to examine the `WHERE` clause because that is where the matching happens. The first triple pattern in the `WHERE` clause

```
?class a owl:Class
```

binds the variable `?class` to the IRI for each class in the ontology. The predicate `a` is "syntactic sugar." In SPARQL, using `a` as a predicate is a short way to use `rdf:type` as the predicate. This is provided because testing for the type of a subject is such a common part of a query. The next line

```
BIND(STRAFTER(STR(?class), 'pattern_examples/') as ?name)
```

uses `STR(?class)` to convert the class IRI to a string. Once we do that, we can perform string operations on it. We then call the `STRAFTER` function with the string for the IRI and the string: `'pattern_examples/'`. For example, the IRI for the class `:Operating_Unit` is https://michaeldebellis.com/pattern_examples/Operating_Unit. Turning this into a string and then finding the part of the string after `'pattern_examples/'` gives us "Operating_Unit". The next expression

```
BIND(REPLACE(?name,"_", " " ) as ?lblname)
```

replaces any underscores in the string with blanks, leaving us with the string "Operating Unit" which is what we want for the label.

The `OPTIONAL` clause checks to see if the class already has a label. If there is an existing label, it will get bound to the variable `?elbl`. Finally, the `FILTER` does the following:

- Makes sure to weed out `owl:Thing` and `owl:Nothing`. You should do this whenever you perform a query that will modify classes because you ***never*** want to mess with those classes. Not only can doing so produce unpredictable outcomes in your ontology, but it may also do so to people who use your ontology. That tends to make you unpopular.
- Makes sure that the label is not the empty string. This can happen if you have classes with different IRIs in the same ontology (which is common). If that happens, you are usually reusing IRIs from ontologies where the entities already have labels. However, if they don't, all you need to do is modify the part of the query that matches for the ontology name to generate labels for those entities (assuming they conform to the standard of separating words with underscores).
- Makes sure that the class doesn't already have a label. Typically, when you work, you will generate labels for everything in your ontology but then later you may add some new entities. You don't want to generate the same label twice. Also, there can be cases where you want to do things for specific entities that don't

exactly match this pattern, such as include copyright or other symbols in a particular label. Checking for an existing label ensures you will only generate labels where they don't already exist.

The other queries in that file work very similarly for individuals, object, and data properties. For the object and data properties, the equivalent of avoiding tampering with `owl:Nothing` or `owl:Thing` is to make sure not to change `owl:topDataProperty` for data properties and `owl:topObjectProperty` for object properties. Although you will hardly ever see them, there are also properties that are the equivalent of `owl:Nothing` for classes. That is, `owl:bottomObjectProperty` and `owl:bottomDataProperty`. These are mostly used for tools such as reasoners. Many tools such as Protégé simply exclude them from the user's view. However, it is still a good idea to make sure not to change them when working with object and data properties just as you don't want to ever change the class `owl:Nothing` so there are checks in the queries for these properties as well.

We won't go through each query because they are essentially the same except that they deal with different entities. However, I encourage you to execute each query on the `Pattern_Examples` ontology without labels, then run the reasoner and notice how the entities are displayed subtly differently now that they have labels.

These queries can be very useful. I use them all the time to generate labels from intuitive IRIs. All you need to do is to replace the string `'pattern_examples/'` with the name of your ontology. You can also do this with IRIs that use CamelCase for classes and individuals and reverseCamelCase (like CamelCase except the first letter is lowercase) for properties. It just takes a bit more work because you need to use a REGEX expression and also because finding and correctly dealing with acronyms requires a bit of extra work when you don't use underscores to delimit breaks between words. For more on this, see the SPARQL label utilities here: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Listing_3-14_SPARQL_Label_Generation_Updates_Patterns.ru.

One final reminder about transferring what you've learned to more sophisticated SPARQL implementations: queries that add data to the ontology using Snap SPARQL don't do it the standard SPARQL way. `INSERT` is the standard way to add new triples to a knowledge graph. Unfortunately, Snap SPARQL doesn't support `INSERT`. If we were using these queries in some other tool with a complete SPARQL implementation, we would use `INSERT` rather than `CONSTRUCT`. However, you can use `CONSTRUCT` with Snap SPARQL, and it will ask you if you want to add the newly constructed triples to the ontology. Typically, you use `CONSTRUCT` when you want to build triples but not add them to the current graph. For example, you may want to create triples using patterns in one knowledge graph, save those to a file, then reload them into another knowledge graph. That is when you typically use `CONSTRUCT`.

Examples of Linked Data with a SPARQL Endpoint Using Wikidata

With that background, let's now use the SPARQL implementation in Wikidata to perform some more interesting queries. A key concept for Linked Data is a SPARQL endpoint. A SPARQL endpoint, of which Wikidata is a highly useful instance, is a web service that accepts SPARQL queries. You can use the `SERVICE` keyword in SPARQL to match patterns in your local knowledge graph with one or more Linked Data endpoints. This is a very powerful tool that puts at your fingertips a huge, distributed graph database with many different domains all across the world.

The most powerful way to use SPARQL endpoints is within a query that can also match data in your local knowledge graph. But it's worth knowing that most Linked Data sites also include a web form where a user can type a SPARQL query and get the results just from that SPARQL endpoint's knowledge graph. This can be a great way to get familiar with a specific endpoint's model and example data before you start integrating it with your own graphs.

The form for Wikidata can be found at https://query.wikidata.org/. And the sample queries can be found at

https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Listings_3-15_to_3-22_Wikidata_SPARQL_Queries.rq.

To run a query, go to https://query.wikidata.org/ and copy and paste it into the window at the top of the form and click the ▶ button toward the bottom of controls on the left. The query in Listing 3-15 retrieves a list of all humans (`wd:Q5`) who were born in New York City, including even those whose birthplace is recorded indirectly (e.g., as a specific hospital or borough contained within the city). It returns each person's Wikidata item, label, description, and number of sitelinks (how many Wikipedia language editions link to them), sorted by popularity (descending number of sitelinks). As with other queries, paste the prefixes once and leave them as you paste each query. Note: the time for these queries varies quite a bit based on your network and how busy Wikidata is. Remember you are no longer querying a small data set but a very large one. Sometimes the following queries can still be fast (1-5 seconds), but other times, they may be slow or you may even get timeout errors.

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX schema: <http://schema.org/>
#Humans born in New York City
SELECT DISTINCT ?item ?itemLabel ?itemDescription ?sitelinks
WHERE {?item wdt:P31 wd:Q5;           #Any instance of a human
             wdt:P19/wdt:P131* wd:Q60; #Who was born in any
# value that is an 'administrative area of' New York City.
    wikibase:sitelinks ?sitelinks.
    SERVICE wikibase:label { bd:serviceParam wikibase:language "
[AUTO_LANGUAGE],mul,en" }}
ORDER BY DESC(?sitelinks)
```

Listing 3-15 SPARQL Query to Find People in Wikidata Who Were Born in NY City

Let's examine this in detail.

```
SELECT DISTINCT ?item ?itemLabel ?itemDescription ?sitelinks
```

The SELECT finds each person's Wikidata entity (`?item`) and includes

- `?itemLabel`: The human-readable name of the person
- `?itemDescription`: A short description
- `?sitelinks`: Number of Wikipedia language editions that reference this entity (used as a proxy for notability)

The `DISTINCT` keyword is often useful in `SELECT` statements since SPARQL can often find the same triple more than once. Using `DISTINCT` will weed out any duplicate triples from the results.

```
WHERE {?item wdt:P31 wd:Q5;
```

The beginning of the WHERE clause filters to instances of human: `wdt:P31` is the property for "instance of" and `wd:Q5` is the Wikidata ID for "human." Note this is an example of opaque IRIs that use codes rather than intuitive names.

```
wdt:P19/wdt:P131* wd:Q60;
```

This is a property path[1] that traverses birthplace (`wdt:P19`) and administrative hierarchy (`wdt:P131`, "located in the administrative territorial entity"). The `*` means zero or more hops up the administrative chain and `wd:Q60` is New York City. This clever pattern matches: direct birthplace = New York City (e.g., `wdt:P19 wd:Q60`) or indirect birthplace is a hospital or borough that's within NYC. The next line

```
wikibase:sitelinks ?sitelinks.
```

retrieves the number of sitelinks (Wikipedia articles across languages) for the item, used here to sort by notability. The next lines are

```
SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],mul,en"}}
```

This is a special Wikidata feature that automatically fetches the best available label and description in your preferred language; `[AUTO_LANGUAGE]` means try the language of the interface or query context; `mul` is the multilingual fallback; `en` ensures you get something in English if the first two options fail.

Remember the `SERVICE` keyword is used to query data from a remote SPARQL endpoint or a special service. It allows you to include results from another data source within your current query. When you use `SERVICE`, you're telling the SPARQL engine: "For this part of the query, go ask another system (or special function) for the results." Wikidata provides built-in virtual services like `wikibase:label`, which isn't a remote endpoint but a shortcut to enrich your query results with useful, human-readable labels and descriptions.

Finally, the last part

```
ORDER BY DESC(?sitelinks)
```

sorts the results in descending order by number of sitelinks, so the most well-known people appear first.

Suppose you wanted to see a list of the Nobel Prize winners in Physics, their birthplaces, and their Wikipedia links. To see this example, go to the Wikidata link and copy and paste the query shown in Listing 3-16 into the Query Text box.

```
SELECT ?winner ?winnerLabel ?birthPlace ?birthPlaceLabel ?wikiPage
WHERE {
  ?winner wdt:P166 wd:Q38104 ; # Won the Nobel Prize in Physics
          wdt:P19 ?birthPlace . # Birthplace
  # Retrieve Wikipedia page link
  OPTIONAL { ?wikiPage schema:about ?winner ; schema:inLanguage "en" ;
schema:isPartOf <https://en.wikipedia.org/>. }
  # Retrieve labels
  SERVICE wikibase:label {bd:serviceParam wikibase:language "en". }}
ORDER BY ?winnerLabel  LIMIT 50
```

Listing 3-16 SPARQL Query to Find Nobel Prize Winners in Physics

This SPARQL query retrieves a list of Nobel Prize winners in Physics, along with the following information: their name `(?winnerLabel)`, their place of birth `(?birthPlaceLabel)`, a link to their English Wikipedia page `(?wikiPage)`, if available. The results are sorted alphabetically by the winner's name and limited to the first 50 matches.

The `OPTIONAL` statement in Listing 3-17 finds the English-language Wikipedia page about the winner if one exists. This needs to be within an `OPTIONAL` clause because if it isn't, then only physicists with Wikipedia pages would be returned and that's not what we want. It uses properties from schema.org, a standard vocabulary used by websites (including Wikipedia) to describe things like articles, people, places, etc.

```
OPTIONAL {?wikiPage schema:about ?winner ;
            schema:inLanguage "en" ;
            schema:isPartOf <https://en.wikipedia.org/> .}
```

Listing 3-17 Excerpt from SPARQL Query to Find the Wikipedia Page of the Nobel Prize Winner if It Exists

schema.org is a widely used vocabulary for describing structured data on the web. It is a great example of the original Semantic Web vision. It was created by major search engines (e.g., Google, Bing, and Yahoo) to help websites provide machine-readable information such as who an article is about, what an event is, or where a business is located.

In the excerpt shown in Listing 3-18, schema.org is used to help locate the English Wikipedia article for each Nobel laureate.

```
schema:about ?winner – the article is about the Wikidata entity
schema:inLanguage "en" – the article is in English
schema:isPartOf <https://en.wikipedia.org/> – it comes from Wikipedia
```

Listing 3-18 Excerpt from SPARQL Query Illustrating Use of schema.org

This lets the query extract not just structured data, but also a link back to human-readable content.

The following small query is especially interesting and illustrates how useful Linked Data can be. It will print out ***all the hospitals in the world*** known to Wikidata and place them on a map of the world that you can zoom in on (Listing 3-19).

```
#Map of hospitals across the world
#defaultView:Map
SELECT DISTINCT *
WHERE {?item wdt:P31/wdt:P279* wd:Q16917;
             wdt:P625 ?geo.}
```

Listing 3-19 SPARQL Query That Finds All Hospitals in the World and Displays Them on the World Map

The results should be displayed on a map at the bottom rather than a table as normal. To best view it, you can use the full-screen icon (a square with a gap in each of the four lines) to the left of the results window. First, you will just see big blotches of red because at the standard zoom level, the individual icons overlap, so it looks like a solid color across much of the map. You can zoom in/out with your mouse wheel, the + or – icons, or the magnifying glass icon to select an area to zoom in on. That will get you down to the state/province and then city level where you can click on icons that represent specific hospitals and get data about that hospital.

You may be wondering how the SPARQL query knew how to use a map view. This is a hack… but a useful one:

```
#defaultView:Map
```

This looks like a comment, but it isn't. This is not part of the SPARQL syntax—it's a "magic comment" used by the Wikidata Query Service that tells WDQS to display results using the Map view, rather than the default table. This only works if the query includes geographic data such as coordinates.

The rest of this query is straightforward. The wildcard `?item` is the subject (an entity in Wikidata); `wdt:P31` = "instance of"; `wdt:P279` = "subclass of"; `wd:Q16917` = hospital.

So `wdt:P31/wdt:P279* wd:Q16917` means: "This entity is an instance of something that is a subclass (or subclass-of-a-subclass, etc.) of hospital." This allows for flexibility—catching not only hospitals directly labelled as such but also more specific types like children's hospitals, military hospitals, teaching hospitals, etc. Note that if Wikidata utilized OWL, you wouldn't need the asterisk because the OWL reasoner knows that the subclass relation is transitive and would propagate all the subclass of subclass relations for you.

Finally, `wdt:P625` = geographic coordinates, so only hospitals with a known location are included. The very last part, `?geo`, captures those coordinates and is what powers the map visualization.

For our last example, we'll use the view magic comment to see another nice view: the graph view in Wikidata. The following query will find links to Lady Ada Lovelace. She was the daughter of the poet Lord Byron and, because she programmed the Analytical Engine of Charles Babbage, is widely recognized as the

world's first programmer. Use the query shown in Listing 3-20 to find her and related objects, then scroll down the screen and mouse on various objects to expand them.

```
#Use the graph view to view people and concepts related to Ada Lovelace
#defaultView:Graph
SELECT ?item ?itemLabel ?itemImage ?value ?valueLabel ?valueImage ?edgeLabel
WHERE {
  BIND(wd:Q7259 AS ?item)  # Ada Lovelace
  ?item ?wdt ?value.
  ?edge a wikibase:Property;
        wikibase:propertyType wikibase:WikibaseItem;
        wikibase:directClaim ?wdt.
  OPTIONAL { ?item wdt:P18 ?itemImage. }
  OPTIONAL { ?value wdt:P18 ?valueImage. }
  SERVICE wikibase:label { bd:serviceParam wikibase:language "
[AUTO_LANGUAGE],mul,en". }}
```

Listing 3-20 SPARQL Query to Find Connections to Ada Lovelace and Display as Graph

If you'd like to try this Graph View query with a different person or concept, it's easy to do. All you need is the Wikidata identifier (also called a Q-ID) for the entity you're interested in. Here's how to find it:

EXERCISE 3-1: WIKIDATA GRAPH QUERY

1. Go to Wikidata: https://www.wikidata.org/.
2. In the search bar, type the name of the person or concept you want to explore (e.g., Alan Turing, Taylor Swift, quantum computing,…).
3. Click on the correct result. The URL will look something like https://www.wikidata.org/wiki/Q7251. The Q-number at the end (Q7251) is the identifier you need. Note that there may be several URLs that are retrieved and what you want is the specific URL that refers to the person or concept you want, not an article that mentions them.
4. Once you have the identifier, go to the Wikidata Query Service with the last query in it, and replace the line `BIND(wd:Q7259 AS ?item) # Ada Lovelace` with `BIND(wd:QXXXXXX AS ?item) # Your new item here.`
5. Substitute your chosen Q-ID for QXXXXXX, run the query, and view the graph.

Advanced SPARQL Queries

In this section, we will explore some of the more advanced features of SPARQL, some of which we already used in the previous section.

One powerful feature in some of the previous queries is the use of SPARQL path operators such as * and /. Path operators allow you to match patterns that follow chains of properties, traverse hierarchies, or explore flexible relationships in a graph. These operators behave like regular expressions over properties and are especially useful when working with deeply nested or recursive structures such as component hierarchies, geographic containment, or organizational trees.

For example, instead of writing multiple queries to find the social network of a person (the people who are social relations of that person's social relations and their social relations), you could append the `*` and use `:has_social_relationship*` to match any person that is a social relation (or social relation of a social relation) of a given person. Similarly, chaining properties with `/` lets you express sequences like "born

in a place that is located in New York City." These operators dramatically increase the expressive power of SPARQL and are widely used in platforms like Wikidata. Table 3-2 is a summary of the path operators with examples.

Table 3-2 SPARQL Path Operators

EXERCISE 3-2: GET ACCESS TO ALLEGROGRAPH

In order to execute some more advanced SPARQL queries, we need a SPARQL engine that implements the complete SPARQL specification. We've already used one of them: the Wikidata Query Service. However, that engine can't load the People Ontology. A tool that is free and easy to use that provides a complete implementation of SPARQL as well as a tool called Gruff to visualize the results of SPARQL queries is AllegroGraph. There are three ways to get access to the free version of AllegroGraph:

1. Use the Cloud Version.
2. Download the Docker container.
3. Download the Tar version and build AllegroGraph on a Linux server.

Using the cloud version is the easiest. You can be up and running in a couple of minutes. The only drawback is if you don't log in to your community cloud account in approximately 24 hours, you will start to get emails saying your cloud account is inactive and any graphs you created will be deleted after a few days of inactivity. I recommend starting with the cloud version, then if you want to do more than just the exercises here, go with the Docker option. The Docker option is easier than the Linux version and almost equivalent; the only differences are that sometimes when testing new features, the Linux version is less buggy than the Docker version, and if you want to work at the Linux shell, it is a bit easier to use Linux itself rather than Docker. To get access to the AllegroGraph Community version cloud:

1. Go to the link https://allegrograph.com/downloads/ .
2. *Select the second option:* **AllegroGraph Cloud Login.**
3. *That will take you to the login page for the AllegroGraph Cloud. Since you don't have an account, select the option* **Don't have an account? Sign up***. Then, create an account and log in, and you are all set.*

To load the Docker container for AllegroGraph, you need to use the instructions at https://franz.com/agraph/support/documentation/docker.html. This will walk you through how to download the (also free) Docker Desktop tool and how to then load the AllegroGraph Docker container.

EXERCISE 3-3: LOADING THE PEOPLE ONTOLOGY INTO ALLEGROGRAPH

This exercise walks you through how to load the People Ontology into the Community Version of AllegroGraph and run the reasoner:

1. Open AllegroGraph and log in using the credentials you created in the previous exercise. This should take you to the root page with a large blue button in the upper right corner that says **+ CREATE REPOSITORY**; click on that. If you are using the cloud version, you'll see a button that says **Create Server**. Select that.
2. In the form that appears, fill in the **Name of new repository*** field with People. Accept the default values for the other options.
3. Click the blue **CREATE & OPEN** button in the lower right corner.

4. That should take you to a screen that describes various options to populate and use your repository. In the column on the left, there should be an option that says **Add, delete & import data**. It should be near the top, right below the Query option. Click on **Add, delete & import data.**
5. That should take you to a screen with the heading **Choose action**. Under that, there will be various options. Click on the option **Import RDF from an uploaded file** (the third one after **Add statements** and **Delete statements**). Click on **Import RDF from an uploaded file.**
6. That should take you to a page with the heading **Select file to import**. Beneath that, there should be a large box with the label **Drag-N-Drop an RDF file or click here to select files**. Click on that box, and that should launch the file locator for your operating system. Use that to navigate to the copy of the People Ontology that you saved on your local hard drive.
7. Beneath that box, there should be a pull-down menu: **Import Options**. Click on the ∨ to expand that option. Change the **Auto-Detect** value you see when you expand that from **Format: Auto-detect** to **Turtle**. When you click on **Auto-detect**, you should get a list of possible options with Turtle near the bottom. Accept all the other default values and select the blue **VERIFY & IMPORT FILE** option.
8. You should see a message that says **Success: 2319 statements were imported successfully!**
9. Next, select **Repository control** from the menu options in the left panel. There will be some default in the search box, probably your username. Enter Manage reasoning in that search field. As you start to type, it will auto-complete and you will see **Manage Reasoning** as a link below the search bar. Click on that.
10. You should see a message that says something like **Success: Generated 2901 triples!**
11. Click on the **Gruff** option near the bottom of the options in the left panel. That should take you into Gruff.

EXERCISE 3-4: SOME MORE COMPLEX SPARQL QUERIES

Now, you can use AllegroGraph to run and visualize queries that won't run in Snap SPARQL. If you have access to other complete SPARQL implementations, e.g., GraphDB from Graphwise or Stardog, those will do as well. Any complete SPARQL implementation can do the following queries. I will be describing how to use Gruff in AllegroGraph, but you can use the same queries and similar visualization tools in many other Semantic Knowledge Graph tools such as Graphwise or Stardog.

Go to the SPARQL queries at https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Advanced_SPARQL_Queries.rq.

Gruff has several different views. To enter a query, you need to be in the query view. To do that, use the menu option: **View➤Query View**.

As always, first paste the prefixes at the top of the file into the Gruff GUI. Then, copy and paste the first SPARQL query:

```
SELECT *
WHERE {:Albert_Einstein ?p :Kurt_Godel}
```

This finds all the properties that connect Einstein and Gödel. Gruff should look like Figure 3-6. Next, try the following query that finds all the nodes in the graph that have Einstein as the subject:

```
SELECT *
```

```
WHERE {:Albert_Einstein ?p ?o}
```

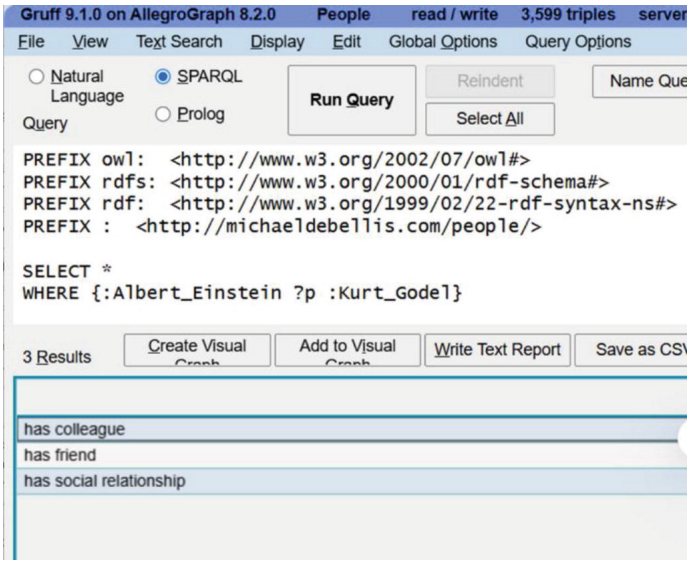


Figure 3-6 SPARQL query in Gruff

Notice how many of the property values are `rdf:type`. This is because Einstein is an instance of the class `:Man`, which has many superclasses such as `:Person`, `:Human`, `:Mammal`, `:Vertebrate`, `:Animal`, and many more. This is an example where the ! path operator is useful. Paste the next query:

```
SELECT *
WHERE {:Albert_Einstein !rdf:type ?o}
```

This query uses the `!rdf:type` path operator to find all triples where `:Albert_Einstein` is the subject, and the predicate is anything except `rdf:type`. However, because property paths abstract over the predicate, the actual predicate is not bound to a variable and cannot be returned so we just get the objects. Path operators are powerful, but they do have this drawback. Another example can be seen with `:has_social_relationship`. First, just get all of Einstein's social relations:

```
SELECT *
WHERE {:Albert_Einstein :has_social_relationship ?person}
```

Now, we will use the * path operator to get the transitive closure of `:has_social_relationship` (all the social relations of Einstein's social relations and all their social relations).

```
SELECT *
WHERE {:Albert_Einstein :has_social_relationship* ?person}
```

Notice that again the information about who is related to whom is missing because we used a path operator. Visualize this in Gruff (select **Create Visual Graph**), and you will see that it is a boring graph; everything just points from Einstein. On the left-hand side of Gruff where it shows the key (the color coding for the edges and nodes), the relation is `:has_social_relationship*` which is a different (computed) relation than `:has_social_relationship`. If we want to see Einstein's social relations and the social relations for each of his social relations, we do **View➤Query View** to go back to the SPARQL query view in Gruff and enter

```
SELECT *
WHERE {:Albert_Einstein :has_social_relationship ?person.
```

```
        ?person :has_social_relationship ?person1.}
```

This isn't the same as using the * path operator because it only gives us one extra level, not all the social relations of the social relations. However, it does preserve the connections. Do **Create Visual Graph** now and you have a more interesting graph. You can also use Gruff to further expand the graph. Right-click on **Alan Turing** and select the first menu option **Display a Linked Node from Menus**. This allows you to see any triple that has Turing as a subject or object. You will be prompted to select whether you want to view triples with Turing as the subject or object. Select **As Subject**. That will bring up a menu of all the predicates that have Turing as a Subject but are not completely displayed. Select the predicate: `:has_social_relationship.` That will bring up the people that Turing has a social relation with who are not yet displayed. Select all. Your graph should now look like Figure 3-7.

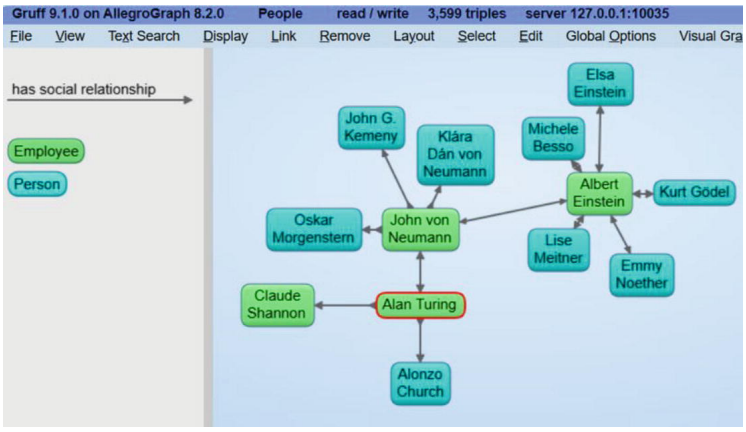


Figure 3-7 Gruff graph after expanding Turing's social relations

Finally, let's use one more path operator; this time the or operator. Select **View➤Query View** and paste and run the query:

```
SELECT *
WHERE {?person :has_child|:has_pet ?living_thing}
```

This lists anyone who has a child or a pet (or both). For example, it includes Confucius and his son Kong Li and Dinsdale Piranha and his hedgehog Spiny Norman.

Conceptual Modeling with OWL

This section describes the basic OWL concepts and their definitions in terms of sets and logic (W3C OWL Working Group, 2012). One of the system development methodology concepts I introduce in this section is the difference between analysis and design. In the old *waterfall model* days, this used to be an important conceptual distinction. People were adamant that the right way to build software was to first get the requirements defined, then build an analysis model, then build a design model and only then write the code. The very first time I was exposed to these ideas, I knew they were wrong from the small amount of programming I had already done. At the time, I couldn't articulate the reasons, and it wasn't until Kent Beck's book on Extreme Programming that someone finally put into words what I always knew was the correct way to build software.

The software world now has embraced iterative, agile development, and as a result, the difference between analysis and design is not that common. It is a concept worth revisiting because one of the things that Semantic Knowledge Graphs deliver is what was the dream of CASE and other tools: to define domains at the analysis level in an executable format and to automate the transformation from the analysis level to

the design and code level. This distinction is especially important in domain modeling because OWL works at the logical or analysis level.

However, as a result, a pure OWL model seldom works in the real world. I have never designed an ontology in Protégé and then used that ontology as the foundation for a Semantic Knowledge Graph with real-world data without requiring that OWL model to be replaced in parts with SPARQL, SHACL, or code in Python or Java. This is an important methodological lesson about designing Semantic Knowledge Graphs that work in the real world with real data integrated with enterprise systems. The most practical approach is to transition from an analysis domain model defined in purely logical terms using OWL and SWRL to a design model that is a combination of OWL with SPARQL, SHACL, and code.

We'll come back to those ideas, but now it is important to give you the required theory you need to know to use OWL. In this section and the rest of the book, I'll be distinguishing between working at the logical analysis level and the graph design level.

Three quick notes about notation I'll be using in this discussion and the rest of the book:

1. When working at the logical level in OWL, we omit prefixes because those are a graph concept rather than a logical concept. For example, when we discuss the Person class in OWL, we use the term `Person`. When we discuss Person in terms of SPARQL and the underlying knowledge graph, we use `:Person`.
2. Protégé uses the labels of entities for definitions and values. Sometimes there is a direct mapping between the user-defined IRI (without a prefix) and the label. For example, `Person`. Other times, especially for properties, the label and IRI are different. When that is the case, Protégé puts the label in single quotes. When you see something like `'is pet of' some Human`, that's what's going on. Protégé is using the labels for the property `:is_pet_of` and the class `:Human`. However, since the label for `:Human` is the same as the IRI, it doesn't bother to put quotes around Human, but since they are different for `:is_pet_of`, Protégé puts quotes around the label string. Protégé can do completion for labels and for IRIs. However, it will only use IRIs when there is no label defined. For most of our examples (and for most ontologies in general), labels will be defined, so when doing a search in Protégé, you should start with a quote unless you are in the early stages of design and haven't yet defined labels for your entities.
3. OWL is based on a formal logic called Description Logic. There is actually a formal language known as Description Logic. I don't use the formal DL notation in this book. I use a language called Manchester Syntax, which is more readable and intuitive and is the language people use to develop virtually all Semantic Knowledge Graphs. However, if you come across the formal language for Description Logic or are just curious, **Appendix B** offers a quick tour and a "cheat sheet" to use to understand the formal language for Description Logic and how it relates to Manchester Syntax. However, in this book, except for that appendix, when I say "Description Logic," I mean the intuitive Manchester Syntax language.

Basic Set Theory

Sets and logic are often presented as complex ideas that require a PhD to understand. Nothing could be further from the truth. We all use sets, and ideas like the intersection and union of a set, countless times a day. It's just that when we use these concepts, they are so natural we aren't even aware of them. The same is true for logic. Many OWL users don't understand the basic theory behind OWL and use it productively. However, it is worthwhile to present the theory, both because understanding it enables you to get the most out of OWL and because the concepts aren't difficult to understand.

Let's begin with sets. A set is an unordered collection of objects called elements. An element is either in a set or not in it. There is no such thing as an element being in a set twice or half in a set. If we add an element to a set that already contains that element, we haven't changed the set. When we describe sets in text, we usually use curly brackets. The set `{Fido, Rex, Buddy}` is the same as the set `{Buddy, Fido, Rex}` or the set `{Fido, Fido, Buddy, Rex, Fido}`. The union of two sets A and B are all the elements that

are in A or in B. The union of `{Buddy, Fido}` and `{Buddy, Fido, Rex}` is `{Buddy, Fido, Rex}`. The intersection of two sets A and B are all the elements that are in A and are also in B. The union of those two sets is `{Buddy, Fido}`. This definition of intersection as A or B and union as A and B will be useful for using OWL to define classes, as we'll show in examples below.

Two sets are disjoint if their intersection must be the empty set, i.e., if there can never be an element in both sets. The two sets we've discussed are not disjoint, because they share the elements `Buddy` and `Fido`.

Sets that aren't disjoint can often be described with the relations subset and superset. Set A is a superset of B if all the elements in B are also in A. Set A is a subset of B if all the elements of A are also in B. The set `{Buddy, Fido, Rex}` is a superset of the set `{Buddy, Fido}` and the set `{Buddy, Fido}` is a subset of `{Buddy, Fido, Rex}`. The subset and superset relations are examples of inverse relations. If A is a superset of B, then B is a subset of A. Note that subset and superset both include cases where all the elements of one set are in the other, i.e., each set is a subset and superset of itself.

A class in OWL is a set. If the class A is a subclass of set B, then A is a superset of B and B is a subset of A. An instance of a class in OWL (also called an individual) is an element of the set that the class is an instance of.

An intuitive way to display set and subset relations is with Venn diagrams. Figure 3-8 shows some Venn diagrams for the classes and instances in the People Ontology that model animals and pets. The diamonds are elements (instances), and the circles are sets (classes). If one circle is completely contained in another circle, that means the contained circle models a subset of the circle it is contained in. In Figure 3-8 `Dog`, `Cat`, and `Hedgehog` are all subclasses (subsets) of `Mammal`. `Halibut` and `Haddock` are subclasses of `Fish`. If two circles don't overlap (e.g., `Dog` and `Cat`), they are disjoint. `Dog`, `Cat`, and `Hedgehog` are disjoint as are `Halibut` and `Haddock`. In a mono hierarchy model (a model that many OOP languages encourage where each class has only one superclass), you would never have any circles that intersect where one isn't completely contained in another. Figure 3-8 shows that the model of animals and pets uses multiple inheritance. Something can be a pet that is also a dog, cat, etc., but not all dogs and cats are pets.

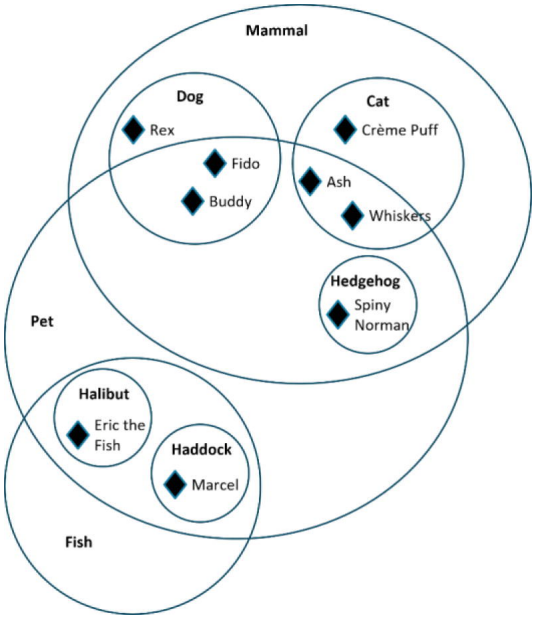


Figure 3-8 Venn diagram of classes and instances of animals

There are two sets that always exist in any OWL model: the empty set and the set that is a superset of every set. In OWL, the empty set is `owl:Nothing`, and the set that is a superset of all sets is called `owl:Thing`. Whenever you view a class hierarchy in Protégé, you will see that all the classes are subclasses of `owl:Thing`. When you run queries in the **DL Query Tab**, you will see that `owl:Nothing` is a subclass of

all classes. If the reasoner ever determines that a class is equivalent to `owl:Nothing`, that means that the class can't have any instances and usually signals an error. It also typically signals an error when the reasoner determines that a class is equivalent to `owl:Thing`. This can happen by having axioms that are too broad and imply that the only class that can satisfy all of them is `owl:Thing`.

We describe a domain in terms of sets, subsets, and relations between sets. You now know almost all the set theory required to use OWL.

A good example of a domain modeled intuitively by sets is the domain of animals. For example, all mammals are animals, and all reptiles are animals, but nothing can be both a reptile and a mammal. That is, animal is a superset of mammal. Reptile and mammal are disjoint. Figure 3-9 shows the animal example class hierarchy that is part of the People Ontology.

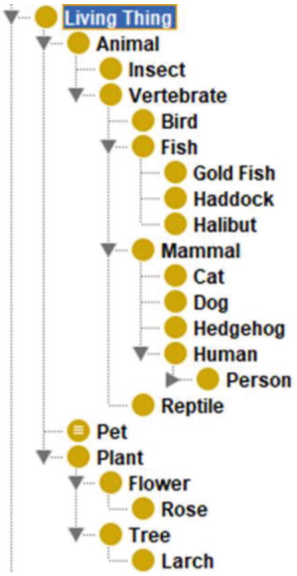


Figure 3-9 Subclass hierarchy example

If you are a programmer, all this should feel very familiar because this is the same semantics that class hierarchies in Object-Oriented Programming use. The difference is that all the mainstream OOP languages lack OWL's ability to analyze your classes and tell you if there is some contradiction in how you've set them up.

A common error of new OOP developers (including me when first learning) was to confuse things like a subclass relation with a part-of or other kind of hierarchical relation. OOP developers also tend to avoid using multiple inheritance. Multiple inheritance is when a class has more than one superclass. For example, the class of "anti-heroes" inherits from both "heroes" and "villains." Many OOP languages like Smalltalk don't even allow multiple inheritance at all, and other languages like C++ probably shouldn't allow it—ask any C++ developer who's wrestled with the diamond problem and virtual inheritance. The complexity of determining which class handles a message in a multiple inheritance hierarchy is often more trouble than it's worth. That's not the case with OWL, of course. OWL models data, not behavior. Multiple inheritance can be a powerful tool to model the fact that there are multiple ways to view a domain. Of course, to paraphrase Spider-Man's Uncle Ben, "with great power comes great ability to mess things up." That is, you want to think carefully about whether and how to use multiple inheritance. This is something we'll cover in detail when we go in depth on how to model a domain: when to create a subclass vs. when to create a sub-part, as well as other modeling techniques.

Another way to think of subclasses and instances is "kind of" and "is a." That is, what it means to say that dog is a subclass of mammal is that a dog is a "kind of" mammal. Likewise, a way to think of what it means for an object to be an instance of a class is "is a." That is, if Fido is an instance of the dog class, then we would

say that "Fido is a dog." This is all very intuitive and is the way we describe and think about the world every day.

Basic Logic

Logic makes people think of proofs and theorems and other intimidating things. But despite all the complex proofs by geniuses such as Turing and Gödel and paradoxes such as Russell's paradox that those of us who love advanced concepts in logic love to discuss, once again as with set theory, logic in OWL is not complex at all—it's just thinking you use every day without being aware of it.

The simplest kind of logic is propositional logic which consists of variables like p and q that can either be true or false and four fundamental operators. In order to define the fundamental operators, we use the term "if and only if," typically signified as iff or with the symbol ≡. This means just what it says. When I say p is true if and only if some condition X is true, it means that if p is true, then X must also be true, and conversely, if X is true, then p must be true, and there are no other possibilities. That is, p and X must have the same value, either both true or both false.

The four logical operators that iff allows us to define are

1. ***And, signified by ∧***: p ∧ q is true iff both p and q are true.
2. ***Or, signified by ∨*:** p ∨ q is true iff either p is true, or q is true. Note this includes cases where both p and q are true. In English, we sometimes say "p or q" when we mean just one of them can be true. Other times we say "p or q" using the logical definition: either one or both. That's the difference between logic and natural language (and between the strengths of semantic technology and of LLMs), logic requires precise definitions, natural language is very imprecise and context dependent.
3. ***If-Then, signified by →*:** p → q is true iff either p is false or both p and q are true. That is, if p is false, it doesn't matter whether q is true or false. This is the only one that may not be completely obvious. But if you think about it for a second, it corresponds to the way we use "if then" in everyday speech. For example, if I say, "if the moon is made of green cheese, then…" the completion doesn't matter because the moon isn't made of green cheese. From a logical standpoint, the sentence will be true, just not very interesting. If-then is often used to stand for a causal relation. For example, "if the Titanic hits an iceberg, then it will sink." "If you drop the ball, then it will fall to the ground." That is why if-then is the foundation for forward chaining rule-based reasoning such as SWRL which we'll explain in a bit.
4. ***Not, signified by ¬*:** ¬p is true iff p is false.

Merging Set Theory with Logic

As we discussed above, intersection of two sets A and B is the equivalent of saying: *any element that is in **A and B***. Union of two sets is the equivalent of saying: *any element that is in either **A or B***. Now is a good time to see Description Logic in action with our simple People Ontology. There are two ways to query an ontology. One is to use logic; the other is to use the underlying graph. Having added the Snap SPARQL plugin to Protégé in **Chapter** 1 gave us one place to do both kinds of queries. We've done some SPARQL (graph) queries in the previous sections. Now we want some basic logic queries to show how `or` and `and` define set union and intersection.

EXERCISE 3-5: SOME DESCRIPTION LOGIC QUERIES IN PROTÉGÉ

Bring up Protégé, load the People Ontology, and navigate to the **DL Query** tab. There should be two sub-tabs on that tab, one for doing logic queries (**DL Query**) and one for doing graph queries (**Snap SPARQL query**). Select the **DL Query** sub-tab. You can enter arbitrary Description Logic expressions into this tab and see all the classes and instances that match. To start, just enter the class Mammal and select **Execute**. Make sure you have the boxes **Instances** and **Subclasses** checked below **Query for** on the right. When you enter Mammal, you should see the results shown in Figure 3-10.

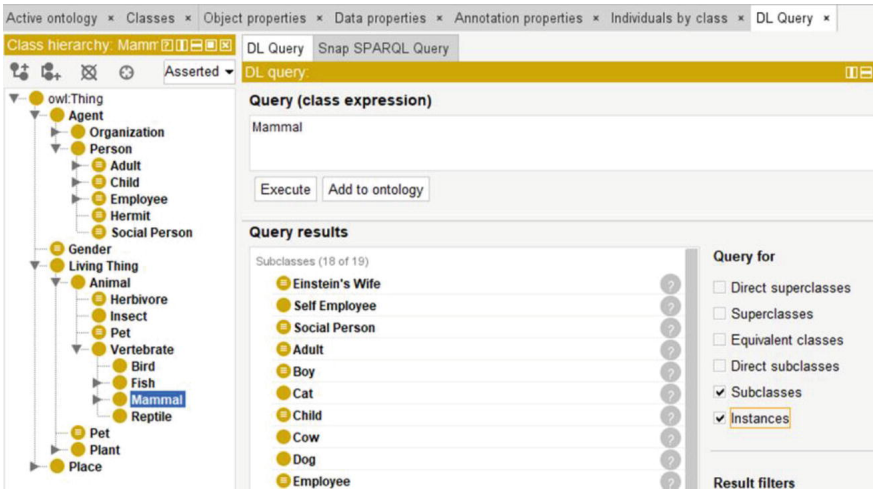


Figure 3-10 Logic query with the DL query tool

There are many classes that are subclasses of Mammal and a lot of instances of those classes as well, so Figure 3-10 doesn't show all your results. To get a smaller set, change Mammal to Dog and select **Execute**.

This should just give you the instances `{Buddy, Fido, Rex}`. Now, let's use *or* to get the union of Dog and Cat. Enter `Dog or Cat` and select **Execute**. That will show you all the Dogs and all the Cats which adds `{Ash, Whiskers, 'Crème Puff'}`. To see the intersection of Dog and Cat, change the or to an and. That should return no instances, because `Dog` and `Cat` are disjoint sets—nothing can be both a `Dog` and a `Cat`. Most of the dogs and cats are pets but not all of them. To see everything that is a `Pet` and either a `Dog` or a `Cat`, enter: `(Dog or Cat) and Pet` and select **Execute**. To find out if there are any Pets that aren't dogs or cats, enter: `(not (Dog or Cat)) and Pet` and select **Execute**. As you will see, there are three pets that aren't a dog or cat: {`Eric the Fish, Spiny Norman, Marcel`}. Double-click on `Marcel` to see that he is an instance of `Haddock` and belongs to `Marcel_Proust`.

Quantification Operators for Defining Sets

Our final theoretical step is to integrate two more operators called quantifiers. A quantifier is a way to make expressions such as "for every element in this set, this logical formula must be true" and "for at least one (aka some) element in this set, this logical formula must be true." The first example is called "for all" or "universal" quantification and is represented by the ∀ symbol. The second example is called "existential" quantification and is represented by the ∃ symbol. The addition of quantification to propositional logic brings us to what's known as First-Order Logic (FOL). FOL is incredibly powerful and interesting, but you don't need to know much about it to use OWL. You don't use FOL in OWL, but rather a less expressive subset known as Description Logic (DL)—which is much simpler and has a more intuitive language.[2]

When using OWL, rather than the mathematical symbols for quantification, we use the keywords `SOME` for existential quantification and `ONLY` for universal quantification. This is because the ways we can use quantification in OWL are more restrictive than in First-Order Logic. The meanings of these quantifiers in OWL are

***SOME (∃)*:** At least one instance of this set must exist.
***ONLY (∀)*:** If this property has values, it must only have elements from a certain set.

The Three Kinds of OWL Classes

An important concept to understand regarding OWL classes is the difference between *necessary conditions* vs. *necessary and sufficient* conditions. Necessary conditions are logical conditions which must be true if

something is an instance of a class. For example, "all pets are nonhuman animals" is a necessary condition for the class `Pet`. If an individual is an instance of `Pet`, then it also must be an instance of `Animal` and not an instance of `Human`.

However, just because some individual is a nonhuman animal doesn't mean it is a `Pet`. Many animals are not pets. Our complete definition for Pet (i.e., the necessary and sufficient conditions for something to be a pet) is

```
Animal and (not (Human)) and ('is pet of' some Human)
```

Thus, if any individual in our ontology satisfies these conditions, the reasoner will infer that it is a `Pet`. If you look at the People Ontology at the class `Pet`, you will see that all the instances of `Pet` were inferred by the reasoner. That is, the user didn't specify that those individuals were pets, but the reasoner was able to infer that they were pets because they satisfied the necessary and sufficient conditions for a Pet.

There are three types of OWL classes:

1. *Primitive Classes*: These are classes that have axioms that provide necessary but not sufficient conditions for an individual to be an instance of the class. The reasoner may be able to infer information about an instance once it is declared as an instance of a primitive class because the axioms indicate conditions that must be true of it. However, it cannot infer whether any individual is an instance of that primitive class because it doesn't have both the necessary and sufficient conditions for that class. Most classes in an ontology are primitive classes.
2. *Defined Classes*: These are classes that have both necessary and sufficient conditions. The reasoner can both infer information about an individual defined to be an instance of a defined class, and it can also infer that an individual is an instance of a defined class even if the user has not explicitly asserted that it is. We have already seen an example of this where the reasoner inferred that instances of `Animal` were instances of `Pet` because they satisfied all the axioms for `Pet` and `Pet` is a defined class.
3. *Anonymous Classes*: Anonymous classes are created by the reasoner as a result of reasoning about various other axioms. For example, if the definition for the domain or range of a property is that it must be an instance of `Dog` or an instance of `Cat`, the reasoner will create an anonymous (i.e., unnamed) class to stand for the union of the classes represented by the Description Logic expression `Dog or Cat`.

Primitive classes show up in the Protégé class hierarchy tab as orange circles. Defined classes are orange circles with three horizontal lines in them (easy to remember because those three lines are also the symbol for if and only if). Anonymous classes don't really show up, but you know they are there. For example, if you see an axiom such as `Girl or Woman`, you know the reasoner created an anonymous class (one that in this case, had it been explicitly created, might have been named `Human_Female_With_Known_Age`).

Relations and OWL Properties

A relation is also a kind of set. However, rather than its elements being individuals, the relation's elements are each tuples (ordered pairs). Unlike a set, the order of the two elements within a tuple matters. Tuples are typically written as `<Mary_Doe, John_Doe>`. The most common kind of relation, and the only kind that OWL supports, is the binary relation. A binary relation has a domain and a range. The domain defines the kind of object that can be the first value of the tuple. The range defines the kind of object that can be the second value. The tuple `<Mary_Doe, John_Doe>`, for example, might be an element in the relation `has_parent`.

The ontology definition of a relation is called a property. This is the difference in OWL between classes and properties. A class is a set of individuals. A property is a set of binary tuples where the two elements of the tuple are individuals (for an object property) or the first element is an individual and the second a simple datatype such as a string (for data properties).

A property's definition includes its domain and range along with various constraints about the property called characteristics. For example, the property `has_parent` at the ontology level is defined as a relation between a `Person` and an `Adult`. An example of a constraint that is defined in the ontology is `has_child` and `has_parent` are inverses. This means that if <`Mary_Doe, John_Doe`> is in the `has_parent` property, then <`John_Doe, Mary_Doe`> must be in the `has_child` property. Or in natural language, if John Doe is the parent of Mary Doe, then Mary Doe is the child of John Doe. Since properties are sets of tuples, `has_child` is a super property to `has_son` and `has_daughter`, because all the tuples in `has_son` and `has_daughter` are in `has_child`.

An example of a property hierarchy in the People Ontology is the hierarchy starting at `has_social_relationship` shown in Figure 3-11. Starting with the `has_daughter` property, as we traverse up the hierarchy, we see that any tuple that is in the sub-property will also be in the super property but not vice versa. For example, <`John_Doe, Mary_Doe`> is in the `has_daughter` property, which means that it is also in the sets defined by the super properties of `has_daughter` which are `has_child`, `has_family_relationship`, and `has_social_relationship`. This corresponds to our intuitive definition of these relations. Since `John_Doe has_daughter Mary_Doe`, then he also `has_child`, `has_family_relationship`, and `has_social_relationship` with `Mary_Doe`. For example, `John_Doe` also `has_child Jim_Doe` but Jim is his son rather than his daughter because the range for `has_daughter` is `Woman or Girl`.

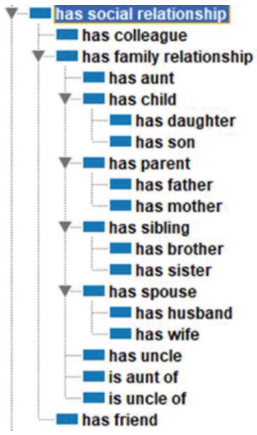


Figure 3-11 Property hierarchy of has social relationship

To see an example of this kind of inference, start Protégé, open the People Ontology, and run the reasoner. Select the **Individuals by class** tab and click on the `Person` class in the **Class hierarchy** view. That should show you all the instances of Person in the **Individuals** view right below it. Select `John_Doe`. In the **Property assertions** view on the right side, you should see all the property values for `John_Doe`, which should look like Figure 3-12. All the values highlighted in yellow are values that were inferred by the reasoner. You can click on the `?` next to any such assertion and get an explanation for how the reasoner inferred this value. For example, select the assertion that `John_Doe has_family_relationship` with `Tom_Doe` and you should see the explanation tells you that `John_Doe has_brother Tom_Doe` and that `has_brother` is a sub-property of `has_sibling` and that `has_sibling` is a sub-property of `has_family_relationship`. You can select any objects that show up as a value in the property assertions and view their values as well. For example, select `Sarah_Doe` which is the value of `has_sibling` and `has_sister`. You will see her property assertions which include that she is the aunt of `Mary_Doe`. Click on the `?` for that inference and you will see an explanation that describes how Sarah Doe is the sister of John Doe and Mary Doe is the child of John Doe and using the SWRL rule:

```
has_child(?p, ?c) ^ has_sister(?p, ?s) -> has_aunt(?c, ?s)
```



Figure 3-12 Property values for John Doe—illustrating reasoner inferences

The reasoner inferred that `Sarah_Doe` is the aunt of `Mary_Doe`.

A practical example of how these relations can be used is a Semantic Knowledge Graph that I developed with Biswanath Dutta for epidemiological data about the COVID-19 pandemic (DeBellis & Dutta, 2022). We had anonymized data that came from many different sources. In some data sets, we knew that one patient was the father of another. In other data sets, we only knew that two patients had some undefined family relation. In yet others, we only knew that two patients had some undefined social relation which could be a family relation or a work or friendship relation. Using OWL properties, we were able to add all the data, at whatever level of specificity was available, and rely upon the reasoner to fill in all correct and relevant inferences.

Three Kinds of Properties

There are three kinds of properties in OWL: object properties, datatype properties, and annotation properties. Object and datatype properties are used to assert values about individuals—not about classes, properties, or other entities. In other words, OWL reserves these properties for use only on instances. This is similar to most Object-Oriented Programming (OOP) languages, where you typically assign values to instances of a class, not to the class itself. That said, many OOP languages include mechanisms to associate data with classes—for example, class variables (also called static fields) in Java allow you to store values that belong to the class as a whole rather than to individual objects.

In OWL, if you want to assert values about a class (such as giving it a description or linking it to an external IRI), you must use annotation properties. Annotation properties are the exception to the rule that properties are only defined on individuals. Annotation properties can be applied to any entity—individuals, classes, properties, even the ontology itself. For example, we've already discussed the `rdfs:label` property to give a class a human-readable name. The `rdfs:label` property is a good example of an annotation property because it is a metadata property. That is primarily what annotation properties are used for. In addition to a "pretty" string, other examples of annotation properties are alternative labels

(skos:altLabel, skos:prefLabel), who created an entity (dct:creator), a textual definition of the entity (skos:definition), and an IRI that is the definition or part of the definition for an entity (rdfs:isDefinedBy), etc. For example, if you are curious about the origins of the pets Eric_the_Fish, Spiny_Norman, and Marcel, you can follow the links of their rdfs:isDefinedBy properties in Protégé.

Most of the annotation properties you use are already defined in commonly used vocabularies: Dublin Core and the Simple Knowledge Organization System (SKOS) and RDFS itself. It is a good idea to get familiar with these vocabularies and to reuse them as much as possible for the annotation properties in your ontologies. Many tools such as Protégé have built-in support for these properties and will provide you with useful functionality by default if you use them. When you open a new Protégé ontology, you will see that it comes with around 15 of the most common annotation properties. In addition, I've created a small ontology I call Basic Reusable Ontology (BRO) that has additional annotation properties that I've found useful on just about any ontology I've developed. That ontology can be found here:

https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/Basic_Reusable_Ontology.ttl

Those annotation properties are useful for most ontologies. When we discuss importing ontologies below, I'll use that as an example of how to import one ontology into another.

The essential theoretical distinction between annotation properties and object and datatype properties is at the graph vs. the logic level. Remember that OWL works at the logic level, while its standard implementation in RDF is at the graph level. At the logical level, we always assert data only on instances of classes. That is what we usually want to do for all kinds of data except metadata, anyway. Following this constraint makes the job of the reasoner more tractable. That is, object and datatype properties can be analyzed by the reasoner and can have additional data added to them by the reasoner. Not so for annotation properties. Annotation properties are completely invisible to the reasoner. That is why they are so useful for metadata—metadata, such as who created an entity or what is its preferred label, are applicable to all entities, i.e., classes and properties as well as instances.

Another advanced technique for associating additional facts with a class is *punning*. Punning allows a single IRI to be interpreted both as a class (or property) and as an individual, depending on the context (W3C, 2012). This enables you to assert object or datatype properties about the class as if it were an individual. This is an attempt to provide a basic meta-object capability in OWL, again similar to class variables in Java. To the reasoner, a punned instance is just another instance. For example, if you have the class Person and you create an instance called Person, that instance is a pun for the class. Protégé and other tools will know that at the graph level they are the same object. However, to the reasoner, the instance Person is just another instance. The fact that it has the same IRI as the class Person is irrelevant to the reasoner because that's a graph issue not a logic issue.

Puns are very useful but in my experience, they sometimes cause interoperability issues. While tools like Protégé can deal with the fact that an IRI is both a class and an instance, the way they deal with it is unique to each tool. That means that sometimes when you transfer files from Protégé to other tools, puns may cause trouble somewhere in the process. Most things you can do with a pun can be done with an annotation property. Thus, when you have the choice, I encourage you to use annotation properties over puns. However, some of the most common ontologies such as SKOS require puns to leverage their full power, so puns often can be very useful, as we will see in **Chapter 12**.

Object and datatype properties are what you use the most when designing an ontology. The difference between the two is their range (the definition of the type of values you can assert in the property). In both cases, the domain (the definition of the type of thing that can have that property as a value) for an object or datatype property is an OWL class. However, for object properties, the range for the property is also a class. Datatype properties have ranges that are other datatypes besides OWL classes such as strings and integers. The most common datatypes are from the XML Schema Definition and are xsd:string, xsd:integer, xsd:decimal, and xsd:dateTime. When you edit the range of a datatype property, Protégé will bring up a long menu of existing datatypes. Those common ones are at the top of the list, and it is best to stick with them.

Also, remember that unlike in OOP, properties are first-class entities; you don't need to associate a property with a class. That is, defining the domain and/or range for a property is always optional. When I first started working in OWL, I came from a software engineering background and assumed (in spite of what experts told me) that it was just a best-practice that a good developer ***always*** defined the domain and range of a property. Then I saw how properties worked with real data and realized that the experts knew what they were talking about. When, why, and how to define domains and ranges (e.g., whether to use OWL or SHACL) is something we will explore in detail in later chapters.

Property Characteristics

There are various characteristics that can be used to enhance the semantics of OWL object properties. These enhancements are only available on object properties because they require a bit of extra overhead that you don't typically need or want for data properties. Consider the example discussed earlier of `has_child` and `has_parent` having inverse properties. When you assert a value on a property with an inverse, the object that is the value of that property will automatically be updated by the reasoner to point to the correct object. For example, see Figure 3-14. If you assert that `John_Doe has_child Susan_Doe`, then the reasoner will assert that `Susan_Doe has_parent John_Doe`. A key virtue of OWL is its ability afforded by characteristics to automatically maintain logic that otherwise would need to be managed in your code. It is rare that you would want to track this kind of information on datatypes other than instances of classes. For example, you seldom would want to keep an inverse property for datatype properties such as `has_age` that points from a particular number to all the instances that have that number as the value for `has_age`. For example, to get the age of everyone in the graph and order it by age from least to greatest, you could use one of the simple SPARQL queries in the **Basic SPARQL Queries** section.

In addition to inverse properties, the other supported characteristics are

- *Transitive Properties*: Transitive properties propagate value from one object to another. The > or < relations are examples of transitive properties. If X > Y and Y > Z, then X > Z. See Figure 3-13. In the People Ontology, the `contains` property is transitive, and you can see several examples of the power of transitive reasoning. When you assert that `British_Columbia contains Vancouver`, the reasoner also infers that `Canada contains Vancouver` (because `Canada contains British Columbia`) and, similarly, `North_America contains Vancouver`.

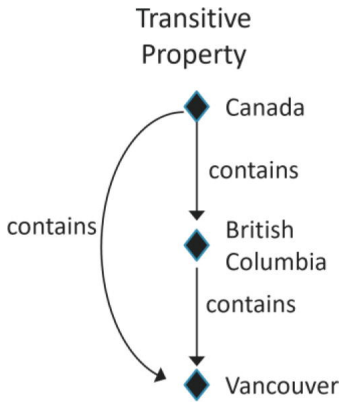


Figure 3-13 Transitive property

- *Functional Properties*: Functional properties must have at most one value for each individual. For example, the `has_age` property is functional. A person can have only one value for their age. This characteristic can also be asserted on data properties. In fact, it is common for data properties such as `has_age` and `birth_date` to be functional.
- *Symmetric Properties*: Symmetric properties point in both directions. For example, `has_sibling` or `has_spouse`. See Figure 3-14.

- *Reflexive Properties*: Reflexive properties are properties that always apply from an individual back to that same individual. See Figure 3-14. Equality is an example of a reflexive property. Any individual is always equal to themselves. OWL provides a property called `owl:sameAs` that is reflexive. This is a way that OWL is different from traditional OOP. Entities are not assumed to be distinct from each other. Reflexive properties always have `owl:Thing` as their domain and range and should be used with great care. They are similar to puns. You need to know what they do because they show up in the Protégé user interface along with these other characteristics; however, unlike other characteristics, they often cause more trouble than they are worth and should only be used by experts.

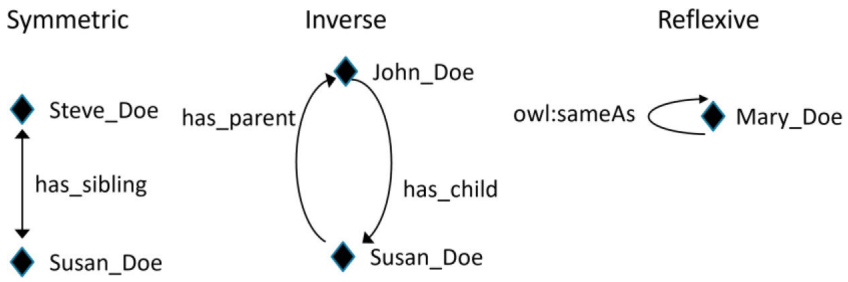


Figure 3-14 Symmetric, inverse, and reflexive properties

- *Super and Sub-properties*: As we've discussed, a super property is more general than its sub-property. For example, `has_parent` is a super property to `has_father`. If `X has_father Y`, then `X has_parent Y`. However, inverses are not necessarily true. Just because `X has_parent Y` does not require that `X has_father Y` (Y could be the mother of X). Super properties are the other exception to the rule that characteristics apply only to object properties. Super properties can apply to data properties as well.

Figure 3-11 shows the property hierarchy of all the properties that are sub-properties of `has_social_relationship`. Now is a good time to bring up Protégé and review an instance of the `Person` class that has inferred property values, such as `Albert_Einstein`. Remember that all reasoner inferences are highlighted in yellow and have a `?` next to them.

EXERCISE 3-6: USE THE REASONER TO EXPLAIN INFERENCES

Now let's use the People Ontology and the explanation feature of the OWL reasoner to understand some of the property inferences it made utilizing the characteristics we've discussed.

Load the People_Ontology into Protégé and start the Pellet reasoner. Bring up `Albert_Einstein` and click on the **?** next to the following values:

```
has_colleague Kurt_Godel
has_employer Institute_For_Advanced_Study
has_spouse Elsa_Einstein
has_friend Emmy_Noether
```

You should be able to validate the explanations for these values:

```
Kurt_Godel has_colleague Albert_Einstein and has_colleague is a symmetric
property.

Institute_For_Advanced_Study is_employer_of Albert_Einstein and
has_employer is the inverse of is_employer_of.

Elsa_Einstein has_husband Albert_Einstein, has_spouse is a super property
of has_husband, and has_spouse is symmetric. Note that has_husband and
```

`has_wife are not inverses. This is to accommodate modeling same-sex marriages.`

`Emmy_Noether has_friend Albert_Einstein and has_friend is symmetric.`

To see an example of a transitive property, find an individual instance of `City` such as `Vancouver`. You will see that all the property values for `Vancouver` are inferred by the reasoner. `British_Columbia contains Vancouver` and `Canada contains British_Columbia` and `North_America contains Canada`. Thus, Vancouver is contained in all of those places because `contains` and its inverse `is_contained_in` are transitive.

Semantic Web Rule Language (SWRL)

The ability to express complex knowledge does not stop with OWL. The Semantic Web Rule Language (SWRL) is a forward-chaining rule language built on top of OWL. Forward chaining means that the reasoner looks for formulas that match the left side of a rule and each time it finds such a pattern it asserts the right side of the rule with the variables on the right side bound to the variables that matched the left side. This can then result in additional rules firing since the rule has added new information to the ontology. This process of a rule firing which adds new data that then causes another rule to fire is called forward chaining.

SWRL allows you to express logical patterns and inferences that go beyond what OWL can define on its own. For example, OWL can define class hierarchies, property restrictions, and even some complex class expressions, but it can't express certain types of multi-individual logic, such as

“If someone is your parent, and your parent has a brother, then that brother is your uncle.”

This kind of rule requires reasoning across relationships between multiple individuals, which is where SWRL excels.

A SWRL rule for this logic would look like

```
has_brother(?parent, ?brother) ^ has_child(?parent, ?child) -> has_uncle(?child, ?brother)
```

This rule can be read as

If a parent has a brother, and that same parent has a child, then the child has that brother as an uncle.

One of the powerful features of SWRL is that its variables and properties automatically imply universal quantification. That is, the reasoner will systematically examine every possible combination of individuals in the knowledge base that satisfies the pattern on the left-hand side of the rule (called the antecedent or body). For example, the expression `has_brother(?parent, ?brother)` means

For any pair of individuals where `?parent` has a `has_brother` relationship to `?brother`, bind those individuals to the variables.

Likewise, `has_child(?parent, ?child)` matches all parent—child relationships, binding `?child` and `?parent`. The rule engine then matches these patterns across all known individuals in the graph.

The arrow (→) separates the condition on the left from the right side of the rule which is known as the consequent. Everything on the left side must be satisfied (matched in the data), and when it is, the rule fires. The reasoner then asserts that everything on the right side of the rule (the consequent) is also true. In our example, the rule will generate new facts by asserting `has_uncle(?child, ?brother)` for each matching pattern it finds.

Because SWRL integrates directly with OWL reasoning, you don’t need separate engines to handle rules vs. ontology axioms. Some OWL reasoners (like Pellet) support SWRL natively, providing a single reasoning pipeline that can combine declarative OWL modeling with procedural rule-based logic.

Defining Classes with Axioms

Now that we've laid the foundation—classes, properties, and the basic structural elements of OWL—we can turn to one of the most powerful and conceptually interesting parts of the language: defining classes using logical axioms.

Unlike traditional object-oriented models, where classes are defined simply by listing their properties and methods, OWL allows us to go further. Using logic—in particular, the Description Logic (DL) subset of First-Order Logic—we can specify exactly what it means to be a member of a class. These logical definitions are expressed as axioms, which enable OWL reasoners to infer class membership, check for inconsistencies, and add implicit relationships in the data. The term axiom is from logic, and it means a logical formula that someone declares or assumes to be true without proof. In OWL, it means the same thing: logical definitions the user declares to be true, and that the reasoner will use to infer additional information, just the way a logician starts with axioms and proves things that follow from those axioms and the rules of logic.

To understand how these restrictions work, let's briefly revisit two fundamental concepts from First-Order Logic and their implementations in Description Logic: existential and universal quantification:

- Existential quantification means that there exists at least one value that satisfies a condition.
- Universal quantification means that all values must satisfy a condition.

Recall that in OWL, these are implemented through the Description Logic operators:

- ***SOME (∃)*:** For existential restrictions—there exists at least one (some) value.
- ***ONLY (∀)*:** For universal restrictions—all values must meet the condition.

In OWL, class definitions typically rely on property-based restrictions—that is, they describe the conditions under which individuals qualify as members of a class based on the properties they have. These restrictions fall into three main categories:

- *Quantifier Restrictions*: Specify that a class must be related to other individuals (or data types) using certain properties. These include existential (`some`) and universal (`only`) restrictions.
- *Value Restrictions*: Specify that a property must have a specific value, range, or set of values.
- *Cardinality Restrictions*: Specify how many values a property must have—minimum, maximum, or exact counts.

These kinds of restrictions can often be combined. Let's take some simple examples from the People Ontology. We want to define the class `Adult` as any `Person` who `has_age` value greater than or equal to 18. Recall that ≡ is our sign for "if and only if." We represent this Defined Class as

```
Adult ≡ Person and ('has age' some xsd:integer[>= 18])
```

This is a quantifier restriction in that it says an Adult must have some value for the `has_age` property that is an `xsd:integer` datatype. It is also a value restriction in that it says the integer must be greater than or equal to 18. If you look at the **Class Hierarchy** tab of the People Ontology, you will see that is the definition for the `Adult` class. Notice the three horizontal stripes in the orange circle for `Adult`. That's Protégé's way to indicate `Adult` is a defined class with both necessary and sufficient (a.k.a. if and only if) conditions.

An example of a quantifier restriction that uses `only` is the definition for the defined class `Herbivore`:

```
Herbivore  ≡ Animal and (eats only Plant)
```

If you look at the `Cow` class, it has a necessary condition:

```
eats only Grass
```

If you also look at the `Cow` class, you will see that the reasoner has inferred that `Cow` is a subclass of `Herbivore` because `Herbivore` is included under **SubClass Of** that is highlighted in yellow which means

it was not defined by the user but inferred by the reasoner. If you click on the **?** next to that inference, you will see an explanation that says

1. Herbivores are animals that eat only plants.
2. Cows only eat grass.
3. Grass is a subclass of plant.

You can see this in another way in the **Classes** tab. Note that on the left side of that tab, there are two views that overlap each other: **Class hierarchy** and **Class hierarchy (inferred)**. The default view (the view on top by default) is **Class hierarchy** which is the user-defined hierarchy. But if you click on the view below it—**Class hierarchy (inferred)**—you see the actual class hierarchy. In that view, you will see `Cow` shows up as a subclass of `Herbivore`. This illustrates a powerful capability of the reasoner and a big difference between OWL and OOP class hierarchies. The reasoner can actually infer that the logical class hierarchy is different than the hierarchy defined by the user as a result of reasoning over the user-defined axioms. For this example, it is a fairly simple change, but for complex classification schemes involving enterprise data, this can be very powerful and a way for the reasoner to help users understand implications of the way they have defined their data that otherwise they might not be aware of. This is also a common standard in the Protégé user interface. You will often see two views overlapping each other, one with the information defined by the user and the other with the information inferred by the reasoner. As you get more accustomed to using Protégé, you will find yourself using the inferred views more and more.

We've seen examples of quantifier restrictions (some and only) and value restrictions (greater than or equal to 18). An example of a defined class that uses a cardinality restriction is `Social_Person` which is defined with the axiom

```
Social_Person ≡ Person and ('has social relationship' min 5 Person)
```

If you look at `Social_Person` in the Class Hierarchy tab, you will see the reasoner has inferred that `Albert_Einstein` and `John_Doe` are instances of `Social_Person` because they both have five or more social relations. Another inference the reasoner has made is that the classes `Social_Person` and `Hermit` are disjoint. No individual can be an instance of both classes. If you select the **?** explanation next to the **Disjoint** declaration for `Hermit` on `Social_Person`, you will see the explanation is that a social person has five or more social relations while a hermit has exactly zero social relations. Obviously, nothing can satisfy both of those axioms, so the reasoner makes the two classes disjoint.

Another type of defined class is an enumerated class. Enumerated classes are excellent ways to model things like codes or tags. We'll discuss this more in **Chapter 8** when we discuss different kinds of buckets and how to model them. In the People Ontology, the class `Gender` is an example of an enumerated class. Enumerated classes are defined by simply listing (enumerating) each individual instance of the class in curly brackets. For example, the definition for `Gender` is `{Male, Female, Non_Binary}`. This and other common ontology patterns will be discussed in **Chapter 8**.

The Open World Assumption (OWA)

The `Hermit` class above is a good starting point to discuss our next important concept: the Open World Assumption (OWA). The term refers to the recognition that the data we're working with is partial and incomplete—and this is not a bug, it's a feature. OWL operates under the Open World Assumption (OWA). This means that just because something is not currently known or stated in the ontology, it does not mean that it is false. This is a major departure from how we are used to working with data, in traditional databases, spreadsheets, or programming languages, where the Closed World Assumption (CWA) reigns: if a value is missing, we assume it doesn't exist. The CWA is used by virtually all other programming and query

languages. For example, Python treats empty collections as False in a Boolean context. The CWA is so common, we take it for granted.

The Open World Assumption is much closer to reality—and how we actually think about things in practice. It accepts that some properties have values, even if not supplied (you believe that I have an age, even though you probably don't know what it is). It accepts that our knowledge is often incomplete.

To see the impact of the Open World Assumption in action, consider our `Person` class with two defined subclasses: `Social_Person` and `Hermit`. As we saw in the previous section, a `Social_Person` is defined as a `Person` with at least five social relationships: `Person and ('has social relationship' min 5 Person)`, while a `Hermit` is defined as someone with none: `Person and ('has social relationship' max 0 Person)`. In the previous section, we saw how the reasoner correctly inferred that both `John_Doe` and `Albert_Einstein` were instances of `Social_Person`, based on their stated relationships. But now consider `Miss_Havisham`, who has no known social relationships in the data. Does that make her a `Hermit`? As you can see in the People Ontology, because of OWL's Open World Assumption, no conclusion is drawn—the reasoner does not assume she has no social relationships, only that we haven't found any yet. More information may still arrive.

The rationale behind the Open World Assumption makes perfect sense when you consider OWL's origins: it was designed to model the World Wide Web, where information is distributed, incomplete, and constantly changing. You never know whether the next HTTP request will uncover new facts. The OWA also applies in the context of Enterprise Data: in real-world organizations, the data you have may come from multiple departments, systems, or third-party sources. There will likely always be relevant data that simply hasn't been integrated into your Data Lake or Data Catalog yet.

This flexibility comes at a cost—you can't use OWL to enforce data integrity constraints, such as "every employee must have exactly one employee ID." That is exactly why the SHACL language was created: to complement OWL by providing Closed World-style validation, enforcing data integrity rules, and generating detailed reports when data fails to conform. SHACL doesn't declare an ontology inconsistent when a constraint is violated; it simply informs you what doesn't match your expectations. What you do about that will depend on what your specific application requires.

In addition to SHACL, you can also use SPARQL to get closed world reasoning. For example, if you wanted to know every `Person` with zero social relationships, you could write the simple SPARQL query (Listing 3-21).

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX :  <https://michaeldebellis.com/people/>
SELECT ?person
WHERE {?person rdf:type :Person.
FILTER NOT EXISTS {?person :has_social_relationship ?otherPerson.}}
```

Listing 3-21 SPARQL Query to Retrieve Any Person with No Social Relations

`FILTER` is a SPARQL expression which evaluates a pattern and filters out any triple that doesn't match that pattern. In this example, `?person rdf:type :Person` will first find all the instances of `Person`. Note this includes any subclass of `Person`. The filter expression will then test each `Person` to see if they match the pattern: `?person :has_social_relationship ?otherPerson`. Since the expression says `NOT EXISTS`, this will filter out anyone who has a value for `has_social_relationship`.

Unfortunately, although the best way to express it, this particular expression illustrates the limitations of Snap SPARQL; this is an example of the SPARQL specification that Snap SPARQL doesn't implement. If you have loaded the People Ontology into AllegroGraph, you can enter this expression and see that `Miss_Havisham` and a few others are returned. However, every good language allows more than one way to say things. If you are only using Protégé, you can see the equivalent information by using the alternative query shown in Listing 3-22 (a form which Snap SPARQL can understand).

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX :  <https://michaeldebellis.com/people/>
SELECT ?person (COUNT(?other) AS ?numRelationships)
WHERE {?person a :Person .
  OPTIONAL {?person :has_social_relationship ?other.}}
GROUP BY ?person
ORDER BY  ?numRelationships
```

Listing 3-22 SPARQL Query to Retrieve Each Person, Count, and List Their Number of Social Relations

This is a simple example of how you can count results in SPARQL. For each person, this will count the number of social relations and return a list with the person and the number of their social relationships and order the list by that number starting with the lowest which of course is zero.

Design Modeling with the Shapes Constraint Language (SHACL)

What OWL and SHACL can model overlaps quite a bit with a few exceptions; the critical difference isn't in ***what*** you can model, but rather in ***how*** that model is best used. In OWL, we focus on conceptual domain models without concern for implementation details. In SHACL, we focus on design issues such as data integrity rules (Knublauch & Kontokostas, 2017). SHACL wasn't part of the original Semantic Web. It was created because when people started to use OWL for real problems, the first thing they tried to do was model data integrity constraints. There are several reasons that OWL is not a good language to do this:

1. *The Open World Assumption*: When we say an employee must have exactly one employee ID, if the employee doesn't have one, it isn't acceptable to say, "Well, it may be out there somewhere." Some of the most important data integrity constraints involve declaring what properties must have a value, and while we can define classes like Hermit that have no values for the `has_social_relationship` property, the reasoner won't make the inference we want due to the OWA.
2. Data Formats: Data integrity constraints sometimes have constraints that go beyond the number of possible or required values and the datatype of such values. We'll see an example of this below with the `employee_id` property. Not only must there be one and only one `employee_id` for each `Employee`, but it must also follow the pattern: `nnn-nn-nnnn`, where n is a digit. That's not a logical constraint, but it still has to be modeled.
3. *Reporting vs. Inconsistency*: At some point, you will see what happens when the reasoner in Protégé finds an inconsistency in your ontology. You can't miss it; usually large parts of your ontology turn red. The reasoner can usually give you useful information on where and what the problem is, but the result is often that much of the ontology turns red. This is because the reasoner is doing logical inferencing—and logical inferencing doesn't like contradictions. All it takes is one inconsistency and the reasoner is no longer useful. This often won't work for real-world data, because real-world data is messy. SHACL generates a report on those data integrity violations without causing the reasoner to mark the ontology as inconsistent.

Point three came as a bit of a surprise to me the first time I took an ontology and deployed it as a knowledge graph with real data. We had worked with the database team and confirmed that certain columns were typed as `xsd:integer, xsd:decimal`, and `xsd:dateTime`. However, in the actual data, we hit cases where numeric fields had values such as "N/A" and "None." Even more so for dates. Date values such as "October 1, 1933," are much more common than `xsd:dateTime` values such as `1933-10-01T00:00:00`.

As a result, I've developed a process that I recommend when developing Semantic Knowledge Graphs which will be discussed in more detail in **Chapter** 4. The core idea lies in the distinction between a conceptual or analysis model and a design model. In this approach to designing Semantic Knowledge

Graphs, we focus first on building a conceptual model in OWL. At this point, we are trying to understand the domain in principle, without any concern for implementation details. We have example data, but it often is fairly simple and doesn't contain the kinds of errors that actual data will. This allows us to focus on the business problem first. That's what our OWL model is: a conceptual model of the domain. Once we have that, we move on to a design model. In this next step, we start using real data with all its faults and peculiarities. Doing so leads to relaxing some parts of our OWL conceptual model and transforming them instead into SHACL constraints. Moving type checking for the range of datatype properties out of our OWL model and into SHACL is one of the most obvious and common examples. However, as we will see, there are many more.

Just to be clear, this is not the return of the waterfall model, which requires all conceptual modeling to be complete before any design can be done. Although people sometimes criticize Agile and other iterative methodologies for ignoring design, that is not the case. Agile has always included conceptual and design models. Somewhat like the difference between OWL and SHACL, the difference isn't ***what*** we model but ***how*** we use the model. In the waterfall method, the goal was to have a complete and consistent analysis model, then a complete and consistent design model and then to write the code. With iterative methodologies, we can build and evolve models simultaneously. What shifts over time is just the balance between them. In early iterations, we focus more on conceptual and design models. In later iterations, design models and code (implementation) become more predominant.

What Is SHACL, Really?

SHACL (Shapes Constraint Language) is a W3C standard designed to express rules about what valid data should look like in an RDF graph. In that sense, it serves a role similar to schemas in relational databases or JSON Schema—not to define what something means logically, but to define what values must be present, what their format is, how many there should be, and so on.

One of the most important things to understand about SHACL is that it's not a separate system—rather, like RDFS and OWL, SHACL is a vocabulary layered on top of RDF. SHACL constraints are written as RDF triples, which means they can be stored, queried, and managed using the same tools you already use for RDF and OWL. When we talk about a "SHACL graph," we're just talking about an RDF graph that contains SHACL terms like `sh:NodeShape`, `sh:property`, and `sh:datatype`.

The central idea in SHACL is called the *shape*. A shape describes what a certain kind of node (often an instance of a class) is expected to look like. You can think of a shape as a format or template: it defines what properties a node should have, what kinds of values those properties should take, and how many such values are allowed. Importantly, SHACL shapes describe structure—not content. SHACL doesn't tell you that Albert Einstein is a physicist or that Berkeley is a university. It tells you that if something is an employee, they must have exactly one employee ID, and that ID must match a certain pattern.

Just like OWL can be used for inference, SHACL can be used for verification. You run a SHACL reasoner against your RDF data, and it produces a report—a list of which data passed or failed the constraints you defined. And unlike OWL reasoning, SHACL is designed to tolerate bad data: it doesn't make the entire ontology unusable if something's wrong. It just tells you what doesn't conform.

The SHACL Editor in Protégé

To keep things simple, we will use a Protégé plugin to see some SHACL examples. However, keep in mind that Protégé is a tool to create OWL models, not to do SPARQL queries or design SHACL constraints. Just as with the **Snap SPARQL** plugin, the **SHACL4Protege** plugin is just meant for training purposes, not for serious work.

EXERCISE 3-7: LOAD THE SHACL4PROTÉGÉ PLUGIN

To add the plugin, start Protégé, load the People Ontology, and run the reasoner as usual. Then do **File➤Check for plugins**. That may take a few seconds, then you will again see a menu showing all of the plugins. Scroll down to **SHACL4Protege Constraint Validator** and click the box to the left, then click the **Install** button at the bottom of the form. You will get a message that says the plugin will be available the

next time you start Protégé. Save your work, close then restart Protégé. Load the People Ontology. You will now have a new tab available. Select **Window➤Tabs➤SHACL Editor**.

That will create a new SHACL tab in Protégé. Navigate to that tab and you will see it has some default SHACL code. However, that example isn't what we want. Go to https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Listing_3-23_to_3-24_Employee_Shape.ttl. You don't need to download this file; you can just copy the code and paste it into the SHACL editor view. There are two shapes in the file. With more robust systems such as the SHACL tools in commercial products such as TopQuadrant, we can process many shapes all at once; however, when using the SHACL Protégé plugin, it is best to handle one shape at a time. To start, copy/paste the prefixes and the first shape into the SHACL EDITOR: `Employee_Shape`.

Then click the **Validate** button in the upper left corner. That will produce a report of the constraint violations. The first two columns in the report—**Severity** and **Source Shape**—aren't important for now. Move those to the left as far as possible so they just appear as three dots. That gives us room for the columns we care about: **Message, Focus Node, Path, and Value**. Expand those columns so you can see the full values. Your screen should look like Figure 3-15.

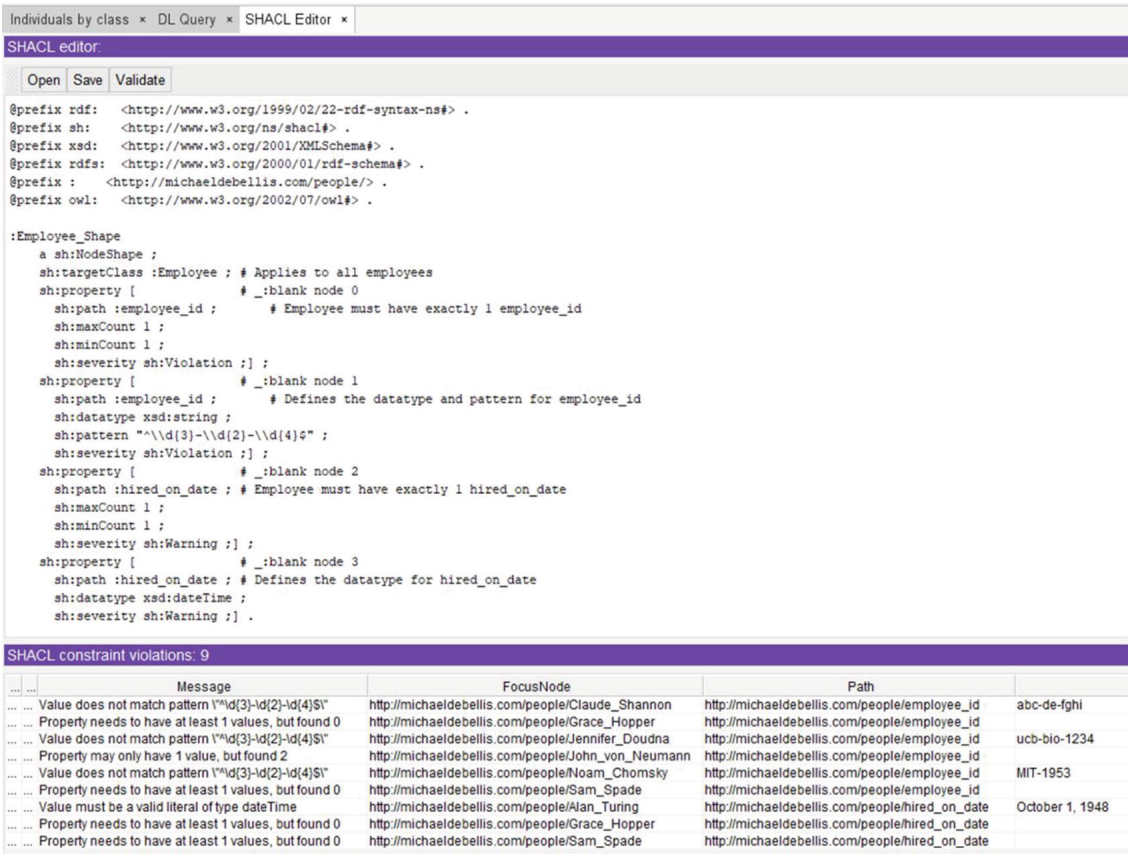


Figure 3-15 The SHACL plugin viewing employee constraints

If the format of the SHACL file looks familiar, that's not a coincidence. If you look at the Turtle code for any OWL ontology, e.g., the People Ontology https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/people_ontology.ttl, you'll see that it and the SHACL file are both written in Turtle. Remember Turtle is a way to save RDF models, and that's what SHACL models are as well.

To really understand how SHACL works, we need to look at the RDF level more carefully. This is a great opportunity to pull back the curtain and see what our models actually look like under the hood. The first triple in the Shape file is

```
:Employee_Shape a sh:NodeShape;
```

Remember the syntactic sugar from a previous SPARQL query. Using `a` as a predicate is shorthand for `rdf:type`. So, this triple says we are creating an object called `:Employee_Shape` that is an instance of `sh:NodeShape`. The classes in SHACL such as `sh:NodeShape` are RDFS classes, not OWL classes. Remember that RDFS is a meta-model layer on top of RDF. Some of the RDFS meta-model such as domain and range of properties are also used by OWL. In other cases, such as Class, an OWL class can do a lot of things that an RDFS class cannot, so we need to create a new meta-model which is what OWL is. Both RDFS and OWL are just additional graphs that define meta-models on top of RDF. The OWL meta-model uses much of the RDFS model such as domain, range, and label. However, it also needs to create some new concepts that go beyond RDFS such as `owl:Class`.

SHACL can define constraints on OWL models, but it doesn't require OWL. This was done so that SHACL could be used for all kinds of systems, e.g., including Linked Data systems that don't want the extra overhead of OWL. Let's continue to analyze the simple SHACL shape. Recall that in SPARQL and Turtle, a semi-colon means that the next line has the same subject as the previous line. The next line in the SHACL shape

```
sh:targetClass :Employee; # Applies to all employees
```

says that the target class for the `:Employee_Shape` is the OWL class `:Employee`. This helps focus the SHACL reasoner. Any properties that we define as part of this shape will be in the domain `:Employee` as a result of this definition. We say "in the domain" because it isn't required that the properties actually have `:Employee` as their defined domain, merely that `:Employee` is within the scope of the kinds of things in the domain. That is, this can include properties that have domains such as `:Person` and `:Agent` because those are both superclasses of `:Employee`.

It makes the most sense to look at the next set of statements together because they span more than one line and introduce an important RDF and SPARQL object: a blank node.

```
sh:property [                 # _:blank node 0
      sh:path :employee_id ; # Employee must have exactly 1 employee_id
      sh:maxCount 1 ;
sh:minCount 1 ;] ;
```

The `[` and `]` characters are used to define nodes in the graph in the definition of another node. We call these blank nodes because they don't have labels and aren't meant to ever be used outside of the context of the node that they are defined in. In this context, this says that an `sh:property` value for `:Employee_Shape` is a blank node that has property values `sh:path :employee_id`, `sh:maxCount 1, sh:minCount 1`, and `sh:severity sh:Violation`.

The first property of the blank node, `sh:path`, defines the property that we are creating constraints for. In this case, `:employee_id`. The next two properties for this blank node describe the actual constraints on `:employee_id`. The minimum and maximum number of values must be one. That is, each employee must have exactly one assigned value. This is an example of a constraint that can be defined in OWL, but due to the OWA, we won't see violations where an `:Employee` has no `:employee_id`, but since SHACL uses the more conventional Closed World Assumption, SHACL will find such violations. The last value for this blank node is `sh:severity sh:Violation`. This defines how serious this error is. In order of severity from least to worst, these are `sh:Info`, `sh:Warning`, and `sh:Violation`. The default if no severity is defined is `sh:Violation`. The `]` means this is the end of the definition of the blank node, and the ; means that the next line will still be defining more properties for the `:Employee_Shape`.

The next lines also define another blank node and another set of constraints for the `:employee_id` property. This time, we want to define the datatype (`xsd:string`) and the format of that string. This is an example of something that can't be easily defined in OWL. The pattern is what's known as a regular

expression or REGEX. It says that the pattern of the string must be `nnn-nn-nnnn` where n is a digit. Regular expressions are used in SPARQL, SHACL, and almost all modern programming languages to describe basic patterns.

The next two property values define constraints for the property `:hired_on_date`. The first says that, like `:employee_id`, `:hired_on_date` must have exactly one value. The second says that the datatype must be an `xsd:dateTime`.

Now that you understand the basics of this SHACL example, let's go through some of the warnings. Make sure you pressed the **Validate** button and that your UI looks like Figure 3-15.

Understanding a SHACL Validation Report

When you run a SHACL validation in Protégé, the results are displayed in a table. Each row represents a constraint violation, and the columns provide details about what went wrong, where, and why. Here's what each column means:

- *Severity*: Indicates how serious the violation is. These don't affect how validation works—they're just labels to help categorize problems. We set the severity in our `:Employee_Shape` to either `sh:Violation` or `sh:Warning`. Notice that the order the SHACL plugin displays the errors corresponds to their severity. It displays the violations before the warnings.
- *Source Shape*: Identifies the SHACL shape (usually a `sh:NodeShape` or `sh:PropertyShape`) that generated the result. This tells you which rule was violated.
- *Message*: A human-readable explanation of what went wrong. SHACL generates a default message that is usually very understandable, so we didn't define a specific message. As you will see, the defaults from SHACL work fine in this example.
- *Focus Node*: The RDF node (usually an individual that is an instance of the class defined as the `sh:targetClass`) that the shape was applied to. This is the data item that failed validation. For example, in the report at the bottom of the plugin, the first focus node is the individual `:Claude_Shannon` who is an instance of `:Employee` because he is employed by `:Bell_Labs`.
- *Path*: The property path that the constraint was checking. If the shape applies to a specific property (like `:employee_id` or `:hired_on_date`), that path is shown here. For node-level constraints, this field may be empty.
- *Value*: The actual value in the data that caused the violation—if one exists. For example, if a string didn't match a pattern or if there were too many values for a property, this will show the offending value. If the problem was a missing required property (e.g., `sh:minCount 1` and no value present), this may be blank.

Looking at the report for this shape, we see the first flag raised is for Claude Shannon. He has an employee ID defined; however, his ID is the string: "abc-de-fghi". This doesn't correspond to the REGEX definition in the `:Employee_Shape`. It has the proper number of characters and dashes, but each character has to be a digit, not an alphabetic character. Similar violations of the constraints for the employee ID property are present: Grace Hopper has no value rather than the required value of exactly one. John von Neumann has two employee ID values, which also violates the cardinality constraint of exactly one.

The `:hired_on_date` violations come next because they were defined with severity `sh:Warning` rather than `sh:Violation`. For example, Alan Turing has the value: October 1, 1948, which is not a valid `xsd:dateTime`. Grace Hopper has no `:hired_on_date` value which violates the cardinality constraint of exactly one.

It is also possible to define constraints that involve two or more property values. These are called node-level constraints. For example, a simple node-level constraint is the second SHACL shape in our example file called `:Employee_Contact_Shape`. It requires that an :`Employee` must have either an `:email` or a `:phone`. That is, they don't need to have both, but they must have at least one (Listing 3-23).

```
:Employee_Contact_Shape
```

```
  a sh:NodeShape ;
  sh:targetClass :Employee ;
  sh:or ([ sh:property [
        sh:path :email ;
        sh:minCount 1 ;
        sh:message "Each employee must have either an email or a phone
number." ;] ]
    [ sh:property [
        sh:path :phone ;
        sh:minCount 1 ;
        sh:message "Each employee must have an email or a phone number."
;]]);
sh:message "Each employee must have either an email or a phone number.".
```

Listing 3-23 The Employee Contact SHACL Shape

This illustrates another feature of SHACL: you don't need to define all the constraints for a class in the same shape. In this case, the logic for contacting an employee is more complex than the other constraints, so we decide to break it out into its own shape for better readability. This shape has one constraint defined with the `sh:or` property. That property takes an RDF list as a value (indicated by parentheses), where each element in the list is a combination of a property name and a blank node that describes a constraint on that property. If you delete the `:Employee_Shape` and paste the `:Employee_Contact_Shape` then select **Validate**, you will get a list of all the employees who have neither an email nor a phone defined.

Viewing the SHACL RDF Graph

In order to reinforce that SHACL constraints are also RDF graphs, we can do a SPARQL query and then visualize the results. This query needs to be run in AllegroGraph or some other tool that has a full implementation of SPARQL. In this case, the SPARQL implementations in Protégé are designed only to work with OWL entities, and SHACL graphs don't utilize OWL. The following query will return all the parts of the `:Employee_Shape` (Listing 3-24).

```
PREFIX : <https://michaeldebellis.com/people/>
SELECT *
WHERE {:Employee_Shape ?p ?o.
?o ?p1 ?o2.}
```

Listing 3-24 SPARQL Code to Match the Graph of the Employee Shape

The first line in the `WHERE` clause returns all the nodes (mostly blank nodes) that are the value of a property of the employee shape. The second line will take each of those nodes and return all the predicates that are associated with it. That will give us a bit more than we need (e.g., it will also return the subclasses of `:Employee` because those are also edges and nodes in the graph) but with some simple pruning in Gruff, and by telling Gruff to lay out the graph as a tree starting at `:Employee_Shape`, we can see the graph in Figure 3-16. All the `:anon` nodes are blank nodes. There are several grey nodes because blank nodes and some other RDF nodes don't need datatypes. For example, the constraint that all employee IDs must be strings with the format `nnn-nn-nnnn` is implemented as the blank node: `:anon294`.

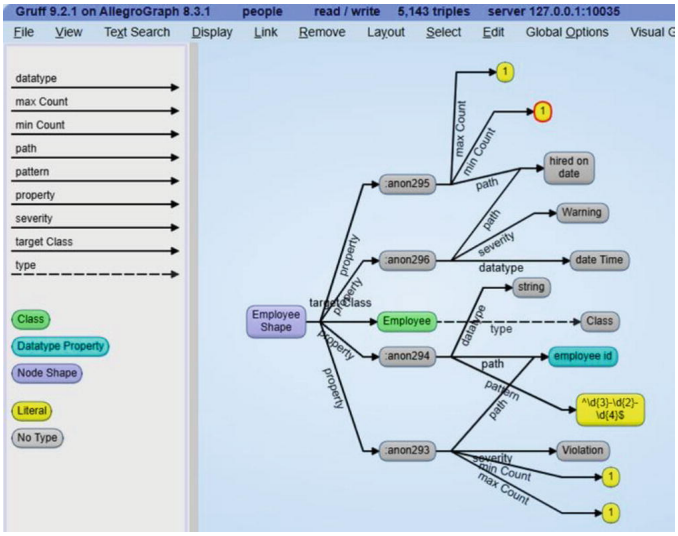


Figure 3-16 The employee shape graph in Gruff

EXERCISE 3-8: THE PIZZA TUTORIAL
Now that you have an overview of the fundamental Semantic Web technologies, it is a great time to go into greater detail on how to use Protégé and other tools to develop Semantic Knowledge Graphs. I recommend you do the Protégé Pizza tutorial at https://www.michaeldebellis.com/post/new-protege-pizza-tutorial. This is a new version of the original tutorial created by the Protégé developers that takes you through hands-on examples of creating an ontology using a simple domain that most people already understand: various kinds of pizzas. Several years ago, I rewrote the tutorial which at the time was no longer consistent with the updated Protégé UI and added several new sections on SPARQL, SWRL, and SHACL. I encourage every reader to take some time now and do the Pizza tutorial. It will give you details about how to use Protégé and other tools that are too detailed for this book but very helpful if you want to use Protégé to build ontologies.

OWL: A Peek Under the Hood

Now that we've gone into how Turtle stores RDF, it is a good time to take a look at what OWL actually looks like when it is saved to a file. Creating and changing ontologies is best done using Protégé and other interactive tools. However, at some point, you are going to need to look at (and even edit) the actual underlying code. This section provides a foundation for what you will need to know in order to do so. You shouldn't want or expect to be able to start editing OWL code directly after reading this section; that's not the intent. Especially for new users, but really for all users, the best way to change an ontology is using a tool such as Protégé. The goal of this section is to provide you with a basic framework for understanding how the logical definitions in Protégé are transformed into code for RDF graphs when Protégé saves a file so that down the road if and when you do encounter an issue that requires you to look at or edit the actual code you have the knowledge to understand the generated code.

Before we begin, it is worth clarifying why I'm explaining this. If I'm telling you that you should almost never do this, why even mention it? The reason is that there are a few good reasons to look at OWL code. These are the main ones:

- *Understanding how to automate changes to the ontology using SPARQL queries*: We'll cover this in more detail in **Chapter** 4, but as you get experience with OWL and SPARQL, you'll find that you can use SPARQL to automate additions and changes to your ontology. This is something I do often. I start by looking at the Turtle code for some example ontology, so that I know that if I want to look at the range of all the data properties, I need to use the `rdf:type owl:DatatypeProperty` and the predicate `rdfs:range` on

the triples that matched the type property. Then, I can use this information, along with documents and spreadsheets, to write SPARQL queries that can generate classes and properties (and of course instances) automatically from documentation.

- *Correcting errors*: It doesn't happen often, but occasionally, you may end up with "junk" in your file. For example, I recently tried an experimental Protégé plugin that added some triples that weren't visible to Protégé, so the only way I could get rid of them was to edit the Turtle file.
- *Making large-scale changes*: You can make almost all changes within Protégé and SPARQL. There are additional tools that we will cover in later chapters such as **Refactor➤Rename multiple entities** that are especially useful for making major changes such as changing many IRIs of your ontology that match a certain pattern. However, every once in a while, you will need to make changes where the easiest way to do it is to just edit the Turtle file using Find/Replace.

Tip If you do need to view/edit Turtle or SPARQL code, I recommend using the free tool: Notepad++. Unlike some other text editors, there is no risk that it will insert invisible characters or otherwise corrupt your file. In addition, Notepad++ will balance brackets, curly brackets, and parentheses. If you put your cursor after a closing parenthesis in Notepad++, the matching open parenthesis will blink red and both the opening and closing parentheses will be red to help you focus on their scope. This helps us understand complex nested expressions such as the class definition we just went through. It also helps us write SPARQL queries which can often have several levels of nesting.

Now to look under the hood:

There are several file formats that can be used to save ontologies and Semantic Knowledge Graphs. These include Turtle, RDF/XML, JSON-LD, and OWL Functional Syntax. Turtle (.ttl) is the most typically used, and that is what we'll use here. You can see the Turtle source code for the People Ontology at https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/people_ontology.ttl. There are two reasons Turtle is one of the best formats:

1. Like RDF/XML, it is one of the most portable formats. One of the formats that is least likely to generate problems when importing/exporting from one tool to another.
2. Of all the formats, Turtle is the format that looks the most like a SPARQL query. Indeed, there are sections of a Turtle file you could simply copy/paste into a SPARQL query (and vice versa) and it would work exactly the same. Learning Turtle helps you understand SPARQL and helps you to write queries to automate changes to the graph.

The ontology begins with all of the required prefixes. After the prefixes, there are a few properties with metadata such as `dct:license` and `skos:definition` that are about the ontology itself. The ontology is also an RDF node that can have properties as well. You can view and edit these values using the **Active Ontology** tab in Protégé.

Protégé prints the Turtle code for a saved ontology in the following order:

1. *Prefixes*: These are the standard prefixes required so the code is readable and each IRI is a short name rather than a long IRI. The format for prefixes in Turtle is that they start with `@prefix`, whereas in SPARQL, they start with `PREFIX`, but except for that, they look and function identically. After the prefixes, the rest of the Turtle file is just a bunch of triples, exactly the same as in SPARQL except in SPARQL there are wildcards for matching while in Turtle there are no wildcards.
2. *Ontology Properties*: The ontology itself is an RDF node and has properties such as who created it, a license, who edited it, comments, etc. These annotation properties can be viewed and edited in the **Active Ontology** tab in Protégé.
3. *Annotation Properties*: A list of all the annotation properties. Whenever you create a new ontology in Protégé, it automatically populates the ontology with the essential annotation properties such as

`rdfs:label`.

4. *Object Properties*: Remember that the reasoner infers a lot of information for object properties and some information for data properties. So often, you won't see things like the domain or range of a property because it can be inferred from a super property.

5. *Data Properties*: All your datatype properties.

6. *Classes*: All your classes along with definitions that define them. Defined classes tend to have more axioms and hence more complex graphs.

7. *General Axioms*: Most axioms are used to define classes and thus are part of the class definition. There are a few axioms (e.g., disjoint axioms) that involve more than one class and are stored here.

8. *Individuals*: For real-world ontologies, individuals are often not stored in the ontology itself, rather the ontology is loaded into a graph database and then some other tool is used to load large amounts of data or data virtualization is used to utilize data in a relational database or other non-graph format. However, what individuals are stored directly with the ontology go in this section.

9. *SWRL Rules*: Any SWRL rules are stored here. SWRL rules are also graphs, and it is possible to automatically transform SWRL into SPARQL rules. This is useful because SWRL rules are very concise and easy to understand, whereas the equivalent SPARQL query will be much larger and more complex. It can be useful to understand the business logic first using SWRL then transform the rules into SPARQL for efficiency.

Within each section, Protégé prints the entities in alphabetic order. Protégé adds comments surrounded by the # character (which, remember, in Turtle and SPARQL indicates a comment) to indicate when it is beginning to display a new type of entity. For example, the object properties begin after the comment shown in Listing 3-25.

```
#################################################################
#    Object Properties
#################################################################
:has_daughter rdf:type owl:ObjectProperty ;
              rdfs:subPropertyOf :has_child ;
              rdfs:range [ rdf:type owl:Class ;
                           owl:unionOf ( :Girl :Woman) ] ;
              rdfs:label "has daughter" .
```

Listing 3-25 Excerpt 1 from Turtle File for the People Ontology

The first line says that `:has_daughter` has `rdf:type owl:ObjectProperty`. Type is the one and only meta-property that is in RDF rather than RDFS or OWL. Type is defined in RDF because the datatype for a node is something that can apply to any node, whether in RDF, RDFS, OWL, or SHACL.

The next line begins with `rdfs:subPropertyOf` and indicates that `:has_daughter` is a sub-property of `:has_child`. Note that there is no domain defined. That is because this property is also a sub-property of `:has_social_relationship` which has the `rdfs:domain` and `rdfs:range` defined as the `:Person` class. Thus, unless we want to further constrain the domain, we don't need to have it in the definition for this or any sub-property; it will be inherited from the super property.

If we do refine the domain or range, we can only make them more rather than less constrained. For example, we refine the `rdfs:range` for `:has_daughter` with an expression that is more restrictive than `:Person`. The next lines that define the `rdfs:range` illustrate an example of an Anonymous Class that is more restrictive than (i.e., a subclass of) the `:Person` class. If you look at this property in Protégé, you will see that the range is `Girl or Woman`. This is stored by Protégé in Turtle as

```
[rdf:type owl:Class ;
  owl:unionOf (:Girl :Woman)] ;
```

Recall that the square brackets are used to define a blank node. The Manchester Syntax statement `Girl or Woman` is implemented by the Turtle expression above that creates a blank node that is an anonymous class that models all human females. That is, the union of the classes `Girl and Woman`. Recall from our discussion of set theory that saying `Girl or Woman` is the equivalent of the union of the two classes, whereas `Girl and Woman` would be the intersection of the two classes. In this example, that intersection would be the empty set because of the way the two classes are defined. A female whose age is 18 or older is a woman; a female less than 18 is a girl. A common error is to define two classes as the domain or range for an object property. This usually doesn't do what you want because when you have two declarations to define a property or a class, they are assumed to be using ***and*** logic not ***or***. A best practice for the domain and range of properties when you have more than one class is to *always* use an expression with the appropriate logical operator. For example, always say `Woman or Girl` when you want the union of the two classes and use `City and State` when you want the intersection of two classes.

The last part of the definition for `:has_daughter` is

```
rdfs:label "has daughter".
```

This label string was generated automatically using the SPARQL queries described in the section: **SPARQL Queries to Generate Labels**.

Next, let's look at an example class, the Adult class. Its definition in Turtle is shown in Listing 3-26.

```
:Adult rdf:type owl:Class ;
   owl:equivalentClass [owl:intersectionOf (:Person
  [rdf:type owl:Restriction ;
  owl:onProperty  :has_age ;
  owl:someValuesFrom [rdf:type rdfs:Datatype ;owl:onDatatype xsd:integer ;
  owl:withRestrictions ([xsd:minInclusive 18])]]) ;
                         rdf:type owl:Class] ;
       rdfs:label "Adult" .
```

Listing 3-26 Excerpt 2 from Turtle File for the People Ontology

The first triple just says that `:Adult` is of type `owl:Class`. The next lines get a bit complex. The easiest way to understand what is going on is to look at the definition for `Adult` in Protégé. First, note that Protégé indicates that `Adult` is a defined class. If you look under Instances in the description of `Adult`, you will see that the reasoner has inferred several individuals are adults. Note: this is not all of the adults in the ontology. It is just the few adults where Protégé can't further determine if they are some subclass of `Adult` such as `Woman`, `Man`, `Girl`, or `Boy`. This is a user interface standard that Protégé uses. Because classes are often instances of many different classes, Protégé only shows instances associated with the most specific classes that they are instances of, and the user understands that the individual is also an instance of all the superclasses of that class. For example, `Emmanuelle_Charpentier` is displayed as an instance of `Woman` and `Employee`. She is also an instance of `Adult`, but the user knows that from the fact that `Woman` is a subclass of `Adult`. This can be confusing at first, but when you think of it, it makes a lot of sense. Taking the other approach and listing all the superclasses would result in a very cluttered user interface. For example, every class is a subclass of `owl:Thing`, and every instance is an instance of `owl:Thing`.

Because `Adult` is a defined class, we expect to see one or more logical axioms in the **Equivalent Class** section of the **Description**. Remember axioms in this section have the logic if and only if (iff, also indicated by an equal sign with an extra line: ≡). For `Adult`, the necessary and sufficient axiom is

```
Person and ('has age' some xsd:integer[>= 18]).
```

This simply means that if an entity is an instance of the `:Person` class and `:has_age some xsd:integer` that is greater than or equal to 18, then they are an `:Adult`.

With that in mind, let's return to the Turtle code for the class. The next line is

```
owl:equivalentClass [owl:intersectionOf (:Person
```

This is the beginning of the graph that makes up the logical definition. It starts with the property `owl:equivalentClass` because `:Adult` is a defined class. Equivalent class means necessary and sufficient conditions. Next, we see an open bracket, so we know what follows is a blank node. In this case, a blank node that defines an anonymous class. This anonymous class is the intersection of two or more class definitions. The class definitions are enclosed in an RDF list signified by parentheses. The first class definition is just the `:Person` class. The second is the expression

```
[rdf:type owl:Restriction ;
  owl:onProperty  :has_age ;
  owl:someValuesFrom [rdf:type rdfs:Datatype ;owl:onDatatype xsd:integer ;
  owl:withRestrictions ([xsd:minInclusive 18])]]);
  rdf:type owl:Class] ;
```

The expression defines an anonymous OWL class, built using a restriction on the `:has_age` property. The restriction is

```
owl:someValuesFrom [rdf:type rdfs:Datatype ;owl:onDatatype xsd:integer ;
  owl:withRestrictions ([xsd:minInclusive 18])]
```

This class includes all individuals that have at least one `:has_age` value that is an integer greater than or equal to 18. Because the class is not given a name, it is anonymous. OWL allows us to nest anonymous classes like this directly inside larger expressions—for example, as part of an intersection used to define `:Adult`. The last part of the restriction

```
rdf:type owl:Class] ;
```

simply says that this entire restriction defines an (anonymous) class. Finally, the last triple defines the `rdfs:label` string value for `:Adult`.

This has been just a basic overview of what the ontology looks like when stored as Turtle because usually you won't work at this level but rather at the logical level via a GUI tool such as Protégé. The idea is to give you an introduction to how these logical concepts are transformed into RDF graphs by Protégé and other tools. Knowing that helps you understand the way you can automatically use SPARQL to transform one type of model to another, e.g., from OWL to SHACL and from SWRL to SPARQL. You are simply transforming one RDF graph into a different RDF graph.

Now that we've gone through the various languages used to define Semantic Knowledge Graphs, the next chapter will describe the process to design them.

Summary

- *An IRI (Internationalized Resource Identifier) is a globally unique identifier used to name resources in the Semantic Web*. IRIs support a wide range of Unicode characters, allowing identifiers in non-English languages. URLs are a kind of IRI meant to link to HTML or other code that can be displayed in a browser. IRIs in general can be much more fine-grained than a document. Every concept, relationship, or instance in a Semantic Knowledge Graph is assigned an IRI. Although it is possible to use different character sets and characters such as blank spaces in IRIs, it is encouraged that for most use cases, you stick with ASCII because using the extended character sets can sometimes cause problems exporting/importing between

different tools. Remember, IRIs are the things developers see; the "pretty names" and names in multiple languages are defined by label annotation properties.

- *RDF (Resource Description Framework) is the foundational data model for the Semantic Web*. It represents information as triples: *Subject ➤ Predicate ➤ Object triples*. RDF provides a simple, flexible structure for linking data across different sources using IRIs. The terms resource and IRI are synonyms. Anything with an IRI is a resource and any resource must have an IRI.
- *RDFS (RDF Schema) extends RDF with basic schema-level vocabulary for defining classes, properties, and hierarchies*. It allows you to declare that certain things are subclasses of others or that properties apply to specific types of resources. RDFS provides the foundation for simple reasoning and classification. OWL uses much of the RDFS vocabulary but there are important differences. For example, an OWL class is different from an RDFS class.
- *Labelled Property Graphs (LPGs) are an alternative graph data model used in tools like Neo4j*. Unlike RDF, LPGs allow properties to be attached directly to other properties and do not utilize global identifiers like IRIs. While more flexible for some use cases, especially those based on mathematical graph theory, LPGs lack the standardization and reasoning capabilities of RDF-based approaches. However, they have many advantages such as strong integration with Hadoop that make them the best technology for certain use cases.
- *Property Graphs and Semantic Knowledge Graphs have fundamental differences; however, the history of software technology shows that what start as competitive models often coalesce into a unified model with the best of both*. There are already many vendors such as Amazon, TopQuadrant, and AllegroGraph that support both models, and the history of technology innovations supports the hypothesis that eventually the technologies will converge.
- *SPARQL (SPARQL Protocol and RDF Query Language) is the standard query language for RDF-based data*. It allows users to retrieve and manipulate data stored in triple form, using pattern matching on graph structures. SPARQL supports querying for classes, properties, values, and relationships. The SPARQL terminology (`SELECT`, `LIMIT`, etc.) is modeled after SQL to make SPARQL easy for SQL developers to learn.
- *OWL (Web Ontology Language) builds on RDF and RDFS to support logical rather than graph models*. It enables the definition of rich class hierarchies, logical constraints, and relationships using Description Logic. OWL ontologies are the backbone of Semantic Knowledge Graphs that support reasoning and inference. OWL uses the Open World Assumption (OWA), which is excellent for reasoning over large, distributed data over the Internet or within a large global organization.
- *The power of OWL is its foundation in logic and set theory*. Some of the most powerful reasoning capabilities in OWL include the ability to create defined classes with logical axioms, the ability to define classes and properties with the same semantics as sets and relations in First-Order Logic, and the ability to define characteristics such as symmetric, transitive, and inverses on properties.
- *SHACL (Shapes Constraint Language) is used to validate RDF data against expected shapes*. Shape is a data science term that means defining a template for a data structure. It allows you to define constraints such as required properties, value types, or cardinality (e.g., "every employee must have exactly one employee ID"). SHACL is critical for ensuring data quality in RDF graphs, including OWL ontologies and Semantic Knowledge Graphs. SHACL uses the more common Closed World Assumption (CWA) for reasoning which is much better suited to data integrity constraints than the OWA. One of the most important things to remember to impress your friends with your understanding of semantic technology is: "OWL is for reasoning, SHACL is for data integrity constraints."
- *OWL and SHACL models are typically implemented as RDF graphs* defined in streaming formats such as Turtle. This makes it possible to use SPARQL to query, build, extend, and transform such models.

Footnotes

1 SPARQL Property paths are described in detail in the next section: "**Advanced SPARQL Queries.**"

2 For those who want to dive deeper into First-Order Logic and how it relates to computer and data science, I recommend the book: (Lewis & Papadimitriou, 1981).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_4

4. The Data Life Cycle

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

This chapter provides an overview of the high-level phases in the data life cycle. The following chapters go into detail on each phase. When we talk about a life cycle for data, it is useful to distinguish between models and processes. Models are the artifacts we produce at different stages of the life cycle: requirements models, conceptual models, design models, and so on. These give us progressively more detail as we move from abstract definitions of business domains to technical execution. Software architects sometimes describe this as going from ***what*** (abstract definitions of the domain and business problems) to ***how*** (technical designs and code).

Figure 4-1 shows the main steps of the data life cycle from a modeling perspective, based on the types of models defined by Strengholt (2023).

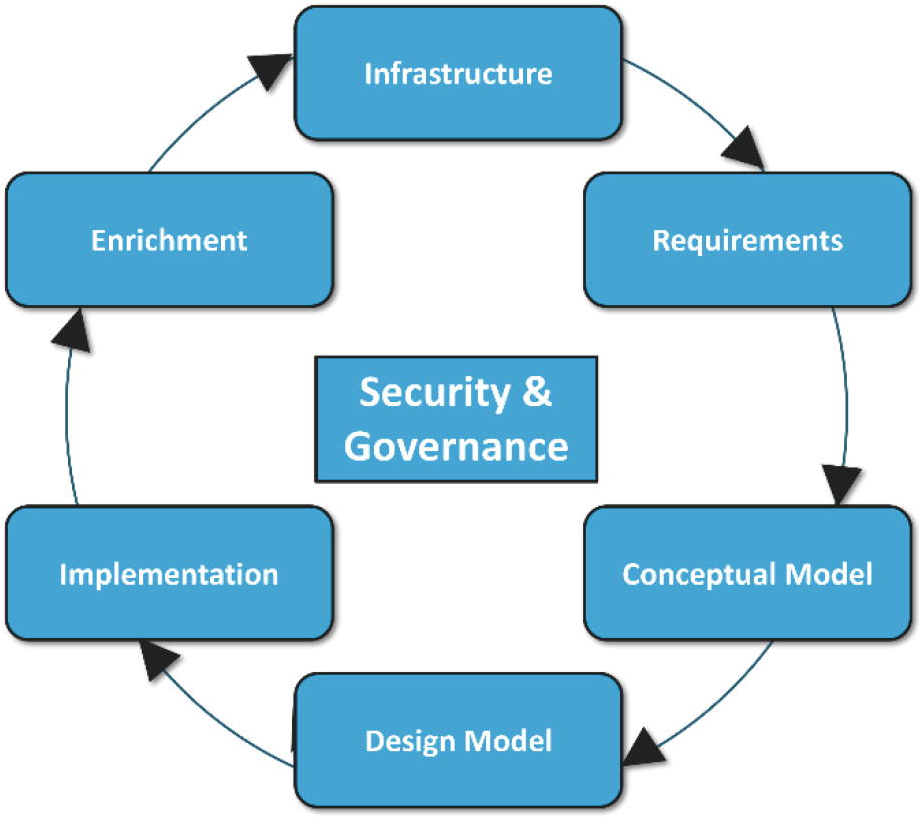


Figure 4-1 The data life cycle

There are many tasks that are pervasive across the life cycle. One example is testing. A major difference between Agile and more traditional development approaches is that Agile does not reserve a separate phase for testing. Testing is continuous. Testing a model differs from testing a model with data, which differs from testing code that uses the model. But in all cases, testing is not a phase—it is an ongoing activity. Another example of a continuous activity is CI/CD (continuous integration and continuous deployment), now the state of the art for teams developing microservices. The older boundaries between system administrators (operations) and developers (implementation) are vanishing.

Figure 4-1 can be deceptive at first glance. It may look like a sequential, non-iterative model arranged in a circle. But this is not what Agile, or any real iterative development, looks like. Dependencies exist: requirements typically come before conceptual modeling, conceptual models usually precede design models, and so on. However, in any

given iteration, the team may be creating several different life-cycle artifacts simultaneously. While one type of artifact tends to precede another, the order is not rigid. Sometimes experience gained while writing code uncovers changes needed in the design architecture or even in the conceptual model.

Figure 4-2 provides an illustration of how work might be distributed in an Agile data process. Precise actual distributions will vary widely. Some projects may focus almost entirely on defining and implementing data governance; others may focus on adding new functionality, with governance work limited to adhering to already established policies. *The essential point is that work occurs at several levels at once*.

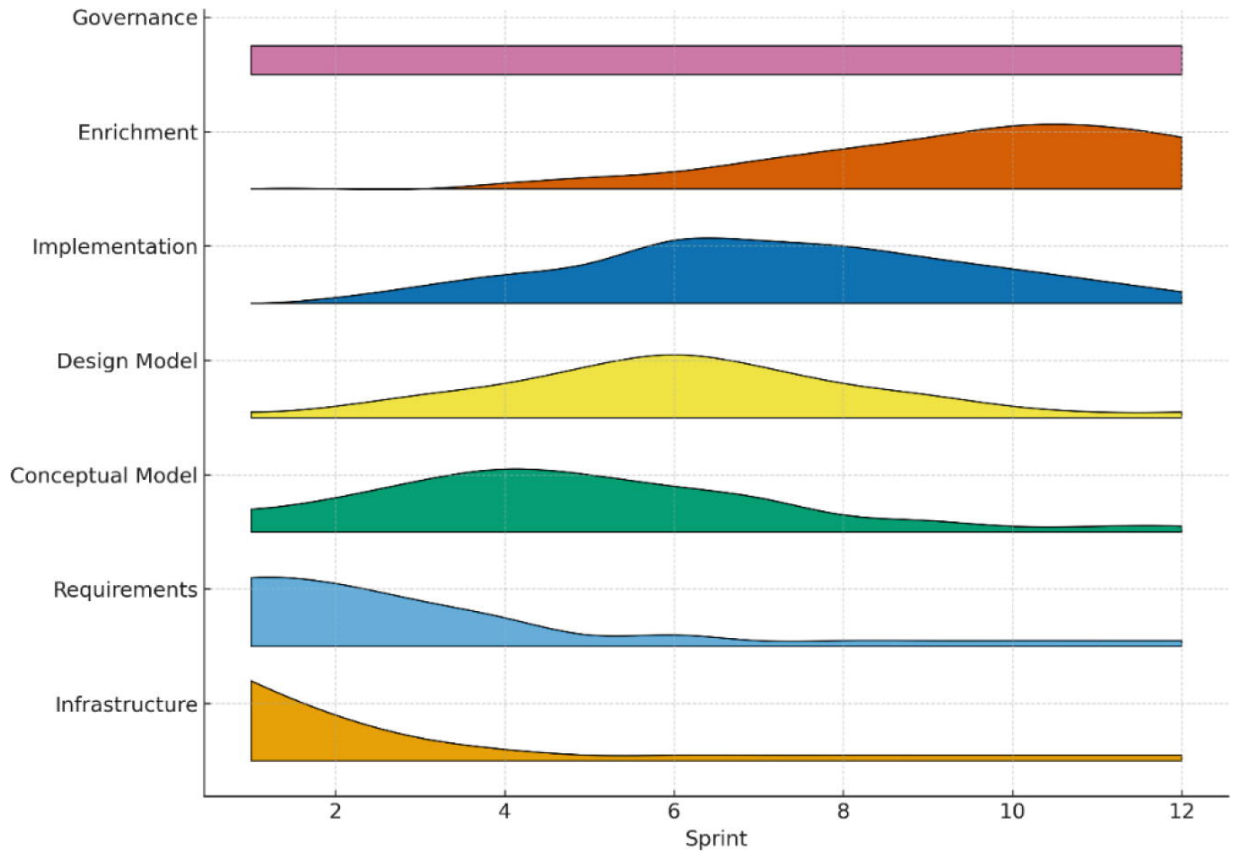


Figure 4-2 Example data life-cycle task distribution

The reason for this is the difference between building a bridge and building software. Building a bridge may have radical and unforeseen

implications for traffic or culture, but bridge builders typically will not know these implications until the bridge is complete. And even if they did discover such issues halfway through construction, the team cannot pivot and redesign the bridge as they go.

Software development is different. One of the main drivers behind iterative processes like Agile is that, throughout the life of a project, business teams gain a deeper understanding of what software can do. This leads them to conceive new ways of solving business problems; ideas they would not have imagined at the outset. Likewise, the technical team gains a better understanding of the business as they work. That business knowledge often suggests new and better technical solutions. The most successful companies are those that can take advantage of this ongoing learning and continuously adapt and exploit new opportunities. This is why one of the core Agile concepts is that "Software development is a knowledge acquisition activity" (Highsmith, 1999).

Overview of Life-Cycle Activities

A key point of Figures 4-1 and 4-2 is understanding that each time segment in the life cycle is a mix of activities. Because particular activities predominate over others at different points in time, it becomes tempting to think of phases as being sequential and entirely about the predominant activity. Making that mistake leads down the path of reinventing the "waterfall" model of software development. Doing so virtually guarantees premature commitments to design decisions. Without look-back and provision for incorporating growing understanding, negative consequences of underinformed decisions readily propagate and grow. Thus, it is important to understand that the dependencies between activities impose only partial ordering. Parallel activity is greater in the midst of a project than at the beginning (when groundwork must be laid) or end (when closure is necessary for the project to terminate).

With this caveat in mind, let's look briefly at each of the key activities.

We start with infrastructure (often called Sprint 0 in Agile) because this is key to standing up and organizing a team. This is where a team defines, installs, and trains itself regarding the infrastructure and tools to be used by the project. These include tools such as GitHub for source code control, Protégé to define ontologies, graph databases, integrated development environments, and data ingestion tools such as Apache Airflow.

This lays the groundwork to define *requirements*, which fundamentally deal with needs and criteria for assessing the quality of potential solutions. These are usually captured in text documents, user stories, use cases, and user interface prototypes. We then create a *conceptual model*, the most abstract model, also called the analysis model. It describes the domain, the business problems we want to solve, and the high-level solutions. These descriptions should be as independent of technical considerations as possible.

Next, we move to the *design model*,[1] where we design the technical solution: which data belongs in a relational database, which belongs in a graph database, the appropriate normal forms for relational structures, expected transaction volumes, data integrity constraints, and so on. We then move to *implementation*, where we deliver data in graph and relational databases and write and test code that operates on the data.

After implementation, we move to *enrichment*. This is one of the core differences between the modern data life cycle and the older software life cycle. In traditional software processes, this step would be called maintenance. But in the modern, data-centric world, our goal is not simply to maintain data as it is; it is to regularly examine it, improve it, and find new ways to make it more useful.

Finally, at the center is *data governance*. Governance sits at the center because it is not just another step—it is the set of rules and practices that shape every other step. Governance answers questions such as: Who owns this data? Who is allowed to change it? How do we

know it is correct? How does it score on the quality metrics we define? Without effective governance, requirements may drift, models may become inconsistent, and enrichment efforts may reduce rather than increase trust in the data.

Governance also defines roles and responsibilities (such as data owners and stewards), establishes consistency and compliance standards, and ensures adherence to legal and organizational requirements for security, privacy, and regulatory oversight. It provides the guardrails for maintaining data quality: validating formats, checking constraints, and ensuring that data products meet stakeholder expectations. We will discuss the concept of a data product in detail with examples in **Chapters 11 and 13**. For now, you can think of a data product as a self-contained unit of data. A data product is functionally similar to a microservice that is designed to be consumed by other systems or teams. However, a microservice concentrates on process and a data product on data.

Done well, governance is more than control. It is an enabler. By ensuring that data is well-described, linked to its sources, and continuously validated, governance lays the foundation for enrichment. Enrichment depends on clear quality standards and shared definitions of what constitutes "bronze," "silver," and "gold" data; without the policies, roles, and validation rules established by governance and enforced by security, these quality tiers would be impossible to define or maintain. Governance creates the conditions under which enrichment can systematically improve the completeness, consistency, and usefulness of data. In this way, governance connects business strategy with day-to-day technical work, ensuring that data not only exists but steadily increases in value.

In the following chapters, each activity in Figure 4-1 has its own chapter. The one exception is enrichment, which is grouped with data governance and security. They are grouped together because all three are concerned with the ongoing health and value of data after it exists in the system. Governance provides the policies and guardrails, security

protects access and integrity, and enrichment improves completeness, quality, and meaning over time.

Agile and the Data Life Cycle

In this book, Agile refers not to a specific branded methodology but to a general way of working: short cycles, continuous feedback, and incremental improvement. It is a process that assumes change will occur and focuses on managing change, not preventing it.

Semantic Knowledge Graph technology fits this mindset naturally. Ontologies can be extended, renamed, or reorganized without the structural rigidity common in other data technologies. SPARQL enables systematic updates and refactoring across large graphs. Modeling, ingestion, validation, and governance therefore coevolve rather than occurring as isolated phases. One of the first things that most new users notice about Protégé is how easy it is to refactor a model with one command and have that change propagated everywhere it is relevant. Agile, in this sense, is about distributing design across the life of the project. OWL and SHACL models grow more precise as real data and use cases reveal what needs improvement.

Finding Your First Project

Understanding the life cycle conceptually is only useful if teams can begin applying it. The first practical question most organizations ask is: where do we start? Before we dive into the details of each life-cycle phase, it's worth discussing an issue that affects every phase: how to choose your first project. This is especially important when introducing a new paradigm like Semantic Knowledge Graphs. Semantic Knowledge Graphs represent a shift in thinking similar to the shift from procedural programming to Object-Oriented Programming. The concepts are simple and intuitive, but they require experience to realize the productivity gains. Teams need practice with small, concrete problems before the modeling style becomes natural.

That means the first project is unlikely to be the most productive or impressive one you'll ever do, and that's fine. Its real purpose is to build shared understanding: learn the tools, the data modeling approach, and the basic life cycle from ontology design through data ingestion and querying to coding, deployment, and enrichment. The second and third projects are where teams begin to see the payoff in reusability, flexibility, and semantic clarity.

For that reason, it's wise to start small. A modest, well-bounded use case lets the team explore the fundamentals without overwhelming complexity. Many popular books on data architectures and related technologies recommend finding a single "killer app" to justify the investment that will radically change the organization. In my experience, success applying new technology is more likely if you instead start with a sequence of small wins that build confidence, demonstrate value, and gradually expand adoption.

Thus, for the first project, I recommend looking for *low-hanging fruit*. Here's an example. When the Internet was just being recognized by the business world, I was trying to develop a new IT practice in a healthcare consulting firm that focused on strategy. Everyone in our small group recognized that the Internet was where we wanted to focus. However, the hesitancy of healthcare CIOs to embrace new technology posed a big challenge. We found an excellent example of low-hanging fruit. Hospitals spent huge amounts of money printing documentation. But their documentation was often obsolete shortly after publication. Once we discovered this, we had some of the easiest sales I've ever experienced. We had a slide deck where we adjusted a few numbers and said: "Right now you're spending $X a year printing and distributing documentation. Give us $Y (which was a small fraction of $X) and we'll build you an Intranet site, put your documentation online, and provide easy tools to update it. You'll never have to worry about printing again and your documentation will always be up to date."

That's an example of low-hanging fruit. The business value was obvious and quantifiable, the technical complexity was low, and as we

built these documentation sites, we gained experience with Internet technology that soon led to more complex applications. These projects also had strategic value for our clients that went beyond the immediate benefits of reducing publication costs. They provided the justification for investing in Intranet infrastructure which would provide many additional benefits beyond the initial simple documentation systems. Semantic Knowledge Graphs are similar: the early projects that work best are those where the value is clear, the scope is manageable, and the solution does not depend on mastering every advanced feature of the technology.

I'll discuss specific applications in **Chapter 14**, and the low-hanging fruit will depend on the specifics of your business and current technical infrastructure. However, here are three examples of projects that can have limited scope and provide obvious business value:

1. *Retrieval-Augmented Generation (RAG)*: RAG systems are a way to provide a customized Large Language Model for your organization and to essentially eliminate black-box reasoning and hallucinations. This is done by augmenting the LLM's internal knowledge with a curated external corpus. Using a knowledge graph to store the corpus and provide additional context for the documents enhances RAG systems even more because in addition to the documents providing the justification for the LLM responses, the knowledge graph also has additional tools to further explore the documents and the domain (DeBellis et al., 2025; DeBellis et al., 2024). All of the major knowledge graph vendors have added the ability to efficiently store vectors that represent the meaning of segments of a text document. Building a RAG system can provide a powerful tool for customer support, knowledge management, and other domains in a relatively short amount of time.
2. *Vocabulary Alignment*: Many organizations maintain multiple controlled vocabularies that describe the same concepts in slightly different ways. In healthcare, for example, clinical systems often rely on SNOMED CT for diagnoses, while billing and reporting systems use ICD. Mapping these vocabularies is essential for

systems use ICD. Mapping these vocabularies is essential for analytics, reporting, and interoperability, yet the task is often done manually or with basic technology such as spreadsheets. SKOS (Simple Knowledge Organization System) provides a lightweight, standards-based way to express these mappings. Each vocabulary becomes a SKOS concept scheme, and terms within each scheme are represented as SKOS concepts. SKOS then provides well-defined properties for alignment. Once these mappings are defined in RDF, they can be queried, validated, and reused across systems. Machine learning tools such as vector embedding models (e.g., Word2Vec) (Mikolov et al., 2013) can also be utilized to automate the process. Although healthcare vocabularies are a well-known example, the same problem appears in many industries. Large financial institutions often have multiple internal taxonomies that must be reconciled after mergers. Manufacturers maintain separate vocabularies for products, materials, and parts acquired from different divisions. In all of these cases, SKOS provides a simple and effective way to unify terminology, reduce ambiguity, and support consistent reporting across the organization.

3. *Semiformal Models*: These are models from the discipline of information architecture such as glossaries, taxonomies, and folksonomies. I call them semiformal models because while they utilize some of the tools of knowledge graphs, they also emphasize informal structures that are intuitive to users. Building such semiformal models with a Semantic Knowledge Graph and the SKOS ontology can be an excellent way to quickly build a model that provides value and also provides the foundation for ontology models (Talisman, 2025). I'll discuss this in detail in **Chapter 12**.

Summary

- *The data life cycle consists of several interconnected phases:* infrastructure, requirements, conceptual modeling, design modeling,

implementation, and enrichment. These phases are governed by continuous practices in data governance and security that shape and constrain every step.

- *These phases do not occur sequentially*. Agile teams refine requirements, adjust models, evolve code, and validate data iteratively in parallel rather than in fixed order.
- *Semantic Knowledge Graph development aligns naturally with this iterative approach*. Ontologies evolve as teams gain insight into the domain, identify inconsistencies in the data, and discover better ways to represent relationships. The graph-based models used by Semantic Knowledge Graphs make automatic refactoring easier than other types of data models.
- *Governance provides ownership, standards, and validation rules that ensure data remains consistent, secure, and trustworthy*. When implemented well, governance enables value creation rather than limiting it.
- *Enrichment relies on the foundation set by governance and security*. Improving data quality, progressing from "bronze" to "silver" to "gold," requires shared definitions, clear metrics, and reliable validation practices.
- *Semantic Knowledge Graphs support continuous refinement*. Their structure makes renaming, restructuring, and extending models far easier than in traditional data technologies, allowing teams to adapt rapidly as understanding deepens.
- *Test-first thinking applies to Semantic Knowledge Graphs* through early creation of example data and SPARQL queries. These tests catch modeling issues, clarify intent, and help prevent unnecessary complexity.
- *Continuous refactoring is essential for maintaining conceptual clarity and internal consistency*. Because Semantic Knowledge Graph technologies are based on RDF and supported by powerful tools like SPARQL, many refactorings can be automated partially or completely.
- *There is a learning curve with Semantic Knowledge Graphs. Your first project may not show dramatic savings* compared to the technologies

you already know, and that's normal. The best chance of success is to find low-hanging fruit that doesn't require a huge team or investment but can demonstrate the value of Semantic Knowledge Graphs and give your team the experience they need to develop systems that will radically transform your organization.

- *Further Reading*: Starting with this chapter, I'm going to leave one or more references in the summary of most chapters to books and papers I think are outstanding on the same topic. For readers interested in the intellectual foundations of iterative development, I recommend *Extreme Programming Explained* by Kent Beck. Although it does not focus specifically on data or semantic technologies, it remains one of the clearest and most concise explanations of Agile principles. Beck's emphasis on feedback, simplicity, incremental design, and delivering tangible business value over technical or philosophical purity strongly aligns with the approach taken throughout this book.

Footnotes

1 I use the term design model rather than Strengholt's term logical model. In the data-architecture tradition Strengholt follows, a "logical model" is a technology-independent but implementation-oriented specification: detailed enough to guide system design yet abstracted from physical storage. However, this book also discusses logical models in the formal-semantic sense used in OWL and Description Logic. Using Strengholt's term here would therefore risk confusion between these two very different meanings of "logical." I adopt design model to preserve his intended meaning while avoiding ambiguity. Strengholt's terminology is appropriate within his tradition; the ambiguity arises because terms such as "logical" and "entity" carry different meanings in different technical communities.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_5

5. Infrastructure and Standards

Michael DeBellis[1]
(1) San Francisco, CA, USA

Designing a Semantic Knowledge Graph requires more than an ontology. It also requires a set of tools and supporting infrastructure that make it possible to build a user interface and other code, persist the knowledge graph efficiently, control the versions of the knowledge graph, validate, document, and evolve the graph over time. In this chapter, we introduce the minimal set of tools that most projects will need, along with the standards that make these tools interoperable. Each of these tools supports one or more stages of the life cycle introduced in **Chapter 4**. Modeling tools support conceptual and design models; triplestores and SPARQL support implementation; SHACL and ROBOT support regression testing; GitHub supports version control with a distributed team. This chapter provides an overview of the core tools used in Semantic Knowledge Graph projects. Table 5-1 shows the minimal tool kit you will need to build a system that utilizes a Semantic Knowledge Graph. For each type of tool, there is an example tool representing it.

Table 5-1 Basic Semantic Knowledge Graph Toolset

Table 5-2 shows tools that are typically required on any medium or large project but are optional for smaller projects. Each tool category is explained at a high level so you can understand its role in the overall workflow. Later chapters cover many of these tools and specific examples in much greater detail. The goal here is simply to show where each tool fits in the development process and to provide an overview of what each tool does, not to provide exhaustive instruction.

Table 5-2 Additional Semantic Knowledge Graph Tools

This chapter does not attempt to compare every tool or endorse specific vendors. Instead, it explains the purpose of each tool category, how it fits into the life cycle, and what capabilities to look for. By the end of the chapter, you should have a clear understanding of the essential infrastructure for building Semantic Knowledge Graphs and how these tools work together to support the data life cycle.

The technologies mentioned here, whether commercial products or open-source libraries, are provided as illustrative examples only. We are not endorsing any particular vendor. Choosing the right stack is a project in itself. There is seldom one tool that is always

best. The decision depends heavily on your specific context: your existing infrastructure, the skills of your team, the systems you are trying to build, the domain, and your nonfunctional requirements (security, performance, usability, compliance, cost, etc.). Having said that, there are a few tool types (e.g., GitHub for collaboration) where there are de facto standards used by most teams. I'll point out those few cases when discussing the relevant tool.

Ontology Modeling

This is our starting point. We need a tool to define the OWL ontology that defines the conceptual model for the knowledge graph. This is where you define classes, properties, example instances, axioms, and rules. This is one of those tools where there is a de facto standard. I'm sure all readers will already know that standard is Protégé.

Protégé has been around for decades, is widely used in academia and industry, and has a large plugin ecosystem (reasoners, visualization, import/export utilities). Protégé also has one of the best support groups I've ever experienced for an open-source tool. In my opinion, the support for Protégé rivals that of some commercial tools. The primary support mechanism is the Protégé user support mailing list.[1] You can sign up to this list using the link in the footnote. Once you subscribe, you can send emails to the list. There is a small but very knowledgeable group of experienced Protégé users who usually respond within one business day with detailed answers and pointers to useful materials.

There are other tools that are also useful. WebProtégé is a hosted version of Protégé. It is also an excellent tool, especially for the early stages of modeling due to its hosted architecture that enables collaborative distributed design. Due to its focus on collaborative design, there are tools in WebProtégé for threaded discussions to resolve questions about alternative designs, names of entities, etc.

VocBench (ART Research Group, 2025) is a web-based, multilingual, role-based editor developed at the University of Rome Tor Vergata. It is used in European projects for SKOS vocabularies, thesauri, and ontologies.

The NeOn tool kit (Open University, 2025) is an Eclipse-based environment for ontology engineering, with support for modularization, reuse, and evolution. For teams that are already using Eclipse, NeOn might be worth investigating. However, their web page describes the project as archived research, so the tool may no longer be maintained.

Graphwise (Graphwise, 2025) and TopQuadrant (TopQuadrant Inc., 2025) have tools that rival Protégé and surpass it in some ways for developing and managing large enterprise ontologies. However, unlike Protégé, these tools are neither free nor open source. They also have additional tools for specific domains. Graphwise is a merger of Ontotext and PoolParty. Ontotext was the vendor for one of the leading graph databases: Graphwise GraphDB. As with all Semantic Knowledge Graph databases, they have tools for visualizing graphs but not tools for creating or changing an ontology that compare with Protégé. PoolParty is the group now in Graphwise that specializes in knowledge management and semiformal models such as taxonomies, and they provide tools similar to Protégé. TopQuadrant focuses on data governance and Data Catalog functions. Both PoolParty and TopQuadrant have excellent tools for managing concurrent ontology design with different stakeholders.

Reusable Vocabularies

One of the most powerful advantages of the Semantic Web standards is they give you access to reusable ontologies called *vocabularies* that were designed to solve common technical and domain-specific problems.

Vocabulary ontologies can be divided into three categories: upper models, technical (horizontal) vocabularies, and industry (vertical) vocabularies. Upper models should be lean because they are meant to be the foundation for all ontologies in an organization. An excellent upper model is the Simplified Upper Level Ontology (SULO) (Dumontier et al., 2025). For business ontologies, the gist upper model from Semantic Arts is an excellent foundation (Blackwood, 2020). For technical vocabularies, always begin with the ontologies from the W3C (W3C, 2026). For industry and domain ontologies, look for ontologies supported by respected organizations such as SNOMED for healthcare and FIBO for financial services.

For our minimal tool kit, I encourage each project to begin by adding metadata properties from common vocabularies that virtually every ontology will require. From the Simple Knowledge Organization System (SKOS) (W3C SKOS Working Group, 2005) import: `skos:altLabel, skos:prefLabel, skos:example, and skos:definition`. From Dublin Core (The Dublin Core™ Metadata Initiative (DCMI), 2024): `dcterms:subject, dcterms:title, dcterms:abstract, dcterms:creator, dcterms:contributor, dcterms:created`. The SKOS label properties offer a lightweight but powerful way to model that the same conceptual entity may have different labels that represent it. The Dublin Core properties also describe minimal metadata (creator, contributor, title, etc.) that all ontologies need. What's more, in both cases, these standards are very widely adopted. Using them means you can take advantage of features in various tools that are defined with these specific properties in mind and that users and developers will be familiar with.

For example, Protégé can be configured to utilize many of these metadata properties automatically using **File➤Preferences➤New entities metadata** to automatically add metadata for each new entity and **File➤Preferences➤Renderer** to configure using additional label properties. All of these properties are annotation properties because they define metadata that can apply to classes and properties as well as instances.

In addition, I have created a GitHub repository with three versions of what I call the Basic Reusable Ontology (BRO) (DeBellis, 2025). I created these ontologies because every time I started a new project, I was importing some common entities that were useful on virtually any project. I decided to just spend a little time and created three vocabularies, starting from the most essential metadata annotation properties, adding a new lightweight upper model called SULO (Dumontier et al., 2025) and a few classes from PROV-O, and then finally a version that has all of Dublin Core, all of PROV-O, SULO, and all the SKOS annotation properties. My goals for creating the BRO vocabularies were:

1. Focus on metadata rather than on an upper semantic model as most upper models do. There is a basic upper semantic model provided by SULO and PROV-O, but the majority of the entities are metadata.

2. Focus on reuse based on my experience with clients. There are ***no new entities*** in any of the BRO vocabularies; they are compilations of entities that experience has shown to be useful.

Semantic Knowledge Graph Databases

Semantic Knowledge Graph databases (often called triplestores) store data using RDF and support the core W3C standards: RDF, RDFS, OWL, SHACL, and SPARQL. Every major triplestore vendor provides some way to get hands-on experience without committing to an enterprise license. This makes it easy to start learning the technology before you have a formal project or budget.

One advantage of the Semantic Web stack is consistency. Because the standards are well defined, RDF and SPARQL behave the same way across compliant systems. You can learn modeling, write queries, and build small examples on any standards-based triplestore. That experience will transfer if you later decide to adopt a different platform. This portability makes early experimentation low risk.

For example, you can easily edit the TBox of an ontology in Protégé and then load the Protégé ontology into a graph database in order to have access to a full implementation of SPARQL and the many additional tools that graph databases have such as programming client libraries and graph visualization.

Most vendors offer a free, community, or evaluation edition. These editions differ in terms and capabilities, but they are usually much more generous than the “free trial” software common in other enterprise IT products. Free knowledge graph database editions usually include all core features. The main limits tend to be the number of triples, the number of concurrent users, or the availability of advanced features such as clustering.

Licenses do vary, and the details matter. Most free editions are intended strictly for evaluation, training, or non-production use. In addition to size or concurrency limits, there may be restrictions on commercial, governmental, or production deployments. Because vendors differ, and because licenses can change over time, always review the terms for the edition you plan to install.

For early work, the free editions are an excellent way to learn the fundamentals. They allow you to

- Install and configure a triplestore locally or in the cloud.
- Practice writing SPARQL queries with a SPARQL engine that supports the complete specification.
- Experiment with ontologies and reasoning.
- Define SHACL shapes and test validation.
- Explore data ingestion patterns and tools.
- Experiment with using Python or Java clients with the graph database.
- Utilize the enhanced features of specific vendors such as graph visualization, ACID transactions, and libraries to integrate with other popular enterprise software such as Kafka.
- Integrate with Large Language Models (LLMs). All the major vendors have extensive capabilities for what is called neurosymbolic databases. That is, integrating support for

vectors that LLMs need (neuro) while also supporting the rich knowledge representation of OWL and SHACL (symbolic).

The goal at the learning stage is to become comfortable with the tools and the modeling approach. Once the team understands the basics and has clear use cases, the decision about which enterprise platform to adopt becomes much easier.

When choosing a system for early exploration, the main considerations are practical ones:

- *Local installation vs. managed service*: Local systems are convenient for learning and quick iteration. Managed services may be a better match if your organization already relies on a particular cloud environment.
- *Limits in free editions*: Most vendors restrict the number of triples, the number of concurrent queries, or certain enterprise features. These limits rarely matter for evaluation and training, but it is important to be aware of them.
- *Team experience and ecosystem fit*: If your team prefers open-source tools or already works in Java, an open-source framework may be a good starting point. If cloud-native tools fit your environment better, a managed RDF service may be easier to adopt.
- *Enterprise concerns come later*: High availability, clustering, and sophisticated security should be considered only after the team has experience and real requirements. Early decisions made without this context are often wrong and sometimes expensive.

Table 5-3 summarizes the free community options offered by several widely used Semantic Knowledge Graph vendors. This is not a product review. It is simply a reference to help you understand what is available for further evaluation and training. Also, note that the details of these licenses change, so be sure to review the actual license for any tool before you begin work.

Table 5-3 Community Versions of Semantic Knowledge Graph Databases

These options make it easy to begin working with Semantic Knowledge Graphs long before you need to make any enterprise commitments. They also allow your team to focus on modeling and SPARQL rather than procurement processes. Once you have hands-on experience, you will be in a much better position to evaluate scalability, performance, operational requirements, and licensing.

With the choice of database in place, the next question is how to query it, make changes to it based on patterns, and add additional data. SPARQL is the most important tool for this.

SPARQL Engine

Protégé includes two SPARQL plugins, but neither supports the full SPARQL 1.1 specification. The SPARQL Protégé plugins are primarily intended as tools for people to learn SPARQL basics, not for serious work. Both neglect to implement significant portions of the SPARQL specification such as `INSERT` and `DELETE`. I prefer Snap SPARQL because there is a hack that allows you to add data to the ontology using `CONSTRUCT`, but Snap SPARQL still has many areas where it is weak. For example, in addition to not being able to delete data (which

is required if you want to change rather than add data), the current version doesn't support a wildcard in the predicate position.

Once you move beyond Protégé and adopt a programming library or graph database intended for application development, SPARQL query and update support almost always comes with it. As an illustration of this point, we'll use the SPARQL engine that is bundled with AllegroGraph and the AllegroGraph Gruff visualization tool along with the AllegroGraph Python client. In later chapters, we also use the Python library RDFLib (RDFLib Open Source Community, 2025), which also includes a complete SPARQL engine.

Knowledge Graph Visualization

Protégé is a great tool for defining the TBox (the schema level) of an ontology. However, to get the most out of a Semantic Knowledge Graph, a graph visualization tool for inspecting the knowledge graph is invaluable. It is an excellent tool to communicate design decisions to users, to visualize knowledge graph models of workflows, causal models, data lineage, taxonomies, and countless other domain-specific graphs. Protégé offers two such tools: OWLViz and OntoGraf. They are useful for certain kinds of use cases. However, the amount of control provided to the user is very limited and they don't scale very well.

Since we use AllegroGraph as our example knowledge graph, we use the AllegroGraph tool for graph visualization called Gruff (Aasman & Cheatham, 2011). Gruff provides countless ways to visualize, lay out, and manipulate knowledge graphs.

We use Gruff to generate many of the images in this book, as well as in additional exercises on the book's GitHub repository Wiki.[2] The way Gruff works is the user initially runs a SPARQL query. The results are returned in a scrollable table. At that point, one can select **Create Visual Graph,** and the results are shown as a graph which can be further manipulated and explored.

Data Ingestion

It is possible to create instances in Protégé one by one. The user interface to do this is cumbersome. That's not an accident. The Protégé developers realized that the most convenient way to add data to an ontology is not by hand but rather to use a data ingestion tool to load data from CSV files, JSON files, or a number of additional available formats. This will be discussed in detail in the chapter on the design model. For our minimal toolset, we utilize the Cellfie tool.[3] The URL in the footnote has a full tutorial on how to use Cellfie. You can invoke Cellfie in Protégé using **Tools➤Create axioms from Excel workbook**. That will launch the Cellfie tool wizard which starts by prompting you to provide an Excel spreadsheet file that contains data. From that point, you can define transformation rules. These rules do three things:

1. They provide ways to transform data in the spreadsheet into appropriate datatypes. For example, for data properties, this would be a datatype such as `xsd:dateTime`. For object properties, that datatype refers to instances of a class such as Person.
2. They provide mappings that guide which properties on an instance will hold the new value from the spreadsheet.

value from the spreadsheet.

3. They provide ways to create additional classes and properties as well. This is one of the interesting differentiators of Cellfie compared to most data ingestion tools. Most data ingestion tools focus only on adding instances and property values for the instance. Cellfie also allows you to automate defining the TBox as well.

Version Control and Collaboration

Modern software and ontology projects almost always rely on version control to keep track of changes. This is one of those tools where there is a de facto standard: Git, created in 2005 by Linus Torvalds (the originator of Linux) to manage the massive, distributed development of the Linux kernel. Git is a distributed version control system (DVCS), meaning every developer has a full local copy of the repository and its history. This is different from older, centralized systems: it makes Git very resilient and flexible.

GitHub is a collaboration platform built on top of Git. While Git is the underlying technology, accessed via a shell, GitHub adds the services teams need to work together: hosting repositories in the cloud, managing permissions, supporting team workflows, and integrating tools like issue tracking, continuous integration, project boards, documentation, and a project Wiki. In addition, GitHub provides a GUI rather than requiring you to work at a shell.

The basic concepts are straightforward:

- *Branches*: Developers can create parallel versions of a repository to try out changes without affecting the main line of development.
- *Merging*: Once changes are tested and approved, a branch can be merged back into the main branch.
- *Pull Requests (PRs):* A GitHub feature for proposing a set of changes, reviewing them, and then merging them into the main project. This is where most collaboration happens in large teams.
- *Issues:* Developers and users can add issues. These can be categorized according to several tags such as requests for new features or bug reports.

The GitHub Desktop client makes using GitHub much easier (especially if you aren't a fan of using command-line interfaces). With the Desktop client, you can do all your work with an intuitive GUI. You can work on your ontology locally in familiar tools, then sync changes back to GitHub with just a few clicks. With the Desktop client, you can create your own copy of the repository on your desktop and use tools like Windows Explorer to do basic file operations such as creating and renaming directories. Then when you push your changes back to the server, they will be reflected in the main repository. This keeps the powerful features of Git and GitHub available without requiring command-line expertise.

Loading Ontology Files from GitHub

Loading ontology files from GitHub can be a bit unintuitive. Protégé and other tools can load ontologies directly from GitHub, but only if the URL points to the *raw* ontology file, rather than the human-readable GitHub webpage. Using the People Ontology as an example, when

you navigate to the file in GitHub, the URL looks like this: https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/people_ontology.ttl.

This is *not* the actual ontology file. It is the *HTML page that displays the file* in the GitHub interface. For example, if you give this URL to Protégé's import wizard, Protégé attempts to parse the HTML as Turtle or RDF/XML and fails, showing "Nothing to import" or a blank summary page along with a Java parsing error in the log.

Many Semantic Knowledge Graph tools allow you to load an ontology directly from a URL. This is convenient because it ensures that the ontology used by your tools, your team, and your automated processes all come from a single authoritative source. When using GitHub, the easiest way to obtain a stable URL for an ontology file is through the *Raw* view.

GitHub presents every file in two ways: a formatted *preview* version and a *raw* version. The preview shows syntax highlighting and metadata; the raw version provides the exact file contents. To load an ontology into tools such as Protégé, GraphDB, Apache Jena, or AllegroGraph, you should always use the Raw version.

The process is simple:

1. Navigate to the file in your GitHub repository.
2. Click the **Raw** button in the upper right corner of the file viewer.
3. When the file opens in your browser, copy the URL from the address bar.
4. Use that URL in the tool you are working with.
5. The Raw URL follows a consistent pattern: `https://raw.githubusercontent.com/<user>/<repo>/<branch>/<path>`[4]

For example, the People Ontology used throughout this book can be loaded with the following URL: https://raw.githubusercontent.com/mdebellis/SemanticKG-Design/main/ontologies/people_ontology.ttl

This URL returns the exact file contents and can be used anywhere a tool expects an ontology IRI or a URL for importing RDF/OWL files.

GitHub occasionally displays long or tokenized URLs when viewing files while logged in, especially if you are the repository owner or a collaborator. These URLs may contain session information or access tokens and are not intended for sharing or long-term use. If you see a URL with a long query string (e.g., ending with `?token=...`), simply click *Raw* again or open the Raw view in a private or incognito browser window. The canonical Raw URL will always have the simpler `raw.githubusercontent.com` structure shown above.

Using Raw URLs ensures that your tools load the most up-to-date version of the ontology and that your team has a consistent reference point. Later chapters show how to integrate these URLs into ontology imports, data ingestion pipelines, validation scripts, and automated testing workflows.

You can copy the URL for the raw file and paste that into any field in Protégé or other tools that prompt you for a URL path. For example, if you do **File➤Open from URL**, you can paste that path into the field that comes up and Protégé will load it. The bottom line is: *whenever you get ontologies from GitHub, always make sure you're using the link to the raw file.*

Programming Language Library and Graph Database Client

I've grouped these two tool categories together because they serve a similar purpose which is how to connect your knowledge graph to a programming language. One of the advantages of OWL is that axioms, SWRL rules, and SHACL constraints can embed various kinds of reasoning (e.g., when setting the value for a property, update that property's inverse) directly in the knowledge graph. This is one of the benefits of data-driven design (McComb, 2019). Logic that in the past would be sprinkled (often inconsistently) across several systems can be modeled and maintained in the knowledge graph.

While this can decrease the amount of code you need to write, you still need to write code for features such as a user interface. There are two ways to do this:

1. Use a graph database client
2. Use a library

All of the graph databases described above have clients for Python and Java. These clients allow you to use those languages to read and change the graph, to define Graphical User Interfaces (GUIs) and transaction boundaries, to connect to LLMs, and to access thousands of other features. This is one of those decisions that is really pretty easy. If you use a graph database, then use the client for your language of choice that comes with it.

However, there are many use cases where you don't need a graph database and can get by with loading and saving data in files. For those situations, you should use a programming language library. This is a library that provides tools to load knowledge graphs into memory from files, run SPARQL queries embedded in functions, access and edit the knowledge graph, etc. One of the most common use cases for a library is for writing small applications like a Protégé plugin. You wouldn't want your users to have to go through the trouble of installing a graph database just to use a plugin for SHACL. There are several excellent libraries in Python and Java. The one I show in the table is RDFLib, which is a Python library and which we'll use for some examples later in the book.

Ontology Documentation Generation

This is another category where there is a de facto standard that I highly encourage you to use. It is an open-source tool called Widoco (Wizard for Documenting Ontologies). Widoco takes an OWL ontology as input and automatically generates a professional-looking website that describes the ontology. The output includes

- A human-readable summary of classes, properties, and individuals.

- Annotations such as labels, definitions, and comments.
- Metadata such as authorship, versioning, and license information.
- Links to the raw ontology file.
- Custom sections you write yourself in Markdown (`.md`) format. This can include an introduction, user guide, examples, and change log. Markdown is a lightweight plain-text format (using # for headings, * for bullet points, etc.) that is widely used on GitHub. Widoco will automatically convert these `.md` files into web pages and add them to the generated documentation site.

This solves a major problem: ontologies are written in formal syntax, but many stakeholders (domain experts, managers, even some developers) need a narrative, browsable version. Widoco provides that "bridge," making it much easier to share your work.

To see Widoco-generated documentation for the People Ontology, go to https://mdebellis.github.io/SemanticKG-Design/peopleOntology/.

If you have issues trying out Widoco and saving files to your local hard drive, see this tip for help: https://github.com/mdebellis/SemanticKG-Design/wiki/WidocoLocalFileTip.

Automatic Testing Tools

Automatic testing tools run tests after changes to your ontology or code. SHACL is in many ways an automated testing tool for knowledge graphs. It defines constraints that the data must conform to and prints a report when property values don't conform to a constraint. ROBOT (Jackson et al., 2021) is also a powerful tool for additional automated testing of Semantic Knowledge Graphs. It was developed for use with the Open Biological and Biomedical Ontology Foundry ontologies, but it can be used on any ontology.

GitHub Actions provides a convenient way to automate testing as part of the repository workflow. A team defines a workflow in a YAML file stored in the repository's "`.github/workflows`" directory and specifies the events that should trigger it. For example, a workflow can run whenever a pull request is opened or updated, when changes are merged into the main branch, at scheduled intervals, or when a developer starts it manually. The workflow can then check out the repository, install the required software, and execute the project's test suite. For a Semantic Knowledge Graph project, these tests might include parsing RDF files, validating data against SHACL shapes, checking ontology consistency with a reasoner, executing SPARQL regression queries, or running unit tests for supporting Python or Java code. Running tests on pull requests identifies problems before changes are merged, while running them again after a merge verifies that the integrated version remains valid. Repositories can also require selected workflow tests to pass before GitHub permits a pull request to be merged.

Refactoring Tools

These are tools that automate stereotypical changes (refactoring) to your ontology and knowledge graph such as generating labels from IRIs, adding language tags to labels, finding and deleting duplicate instances, etc. For our example tool set, we'll just use a library of

SPARQL updates that I've created and gathered from other developers.[5] In addition, especially for people working in the Open Biological and Biomedical Ontology (OBO) Foundry, ROBOT can also do refactoring.

Anti-pattern Analysis

Patterns are common good designs that developers reuse. These first gained prominence during the adoption of Object-Oriented Programming for industry via books such as Martin Fowler's *Analysis Patterns: Reusable Object Models* (Fowler, 1997). The goal was to enable reuse at the model level rather than at the code level. An anti-pattern is just the opposite: a pattern that defines a common ***bad*** practice to be avoided. Anti-pattern analysis tools look for anti-patterns and may suggest ways to refactor them. An example that C programmers will be familiar with is Lint.

There are also anti-patterns in ontology design. Many of these will be discovered by the reasoner, but many will not. The OOPS! (Ontology Pitfall Scanner) tool[6] is useful to find potential anti-patterns (Poveda-Villalón et al., 2014). OOPS! is to ontologies what Lint is to C: a static checker that scans your model and warns about potential pitfalls or anti-patterns, leaving it up to you whether to act on them. This can be especially useful for those who are new to designing OWL models. It is also possible to create SPARQL queries that look for anti-patterns. Many organizations that build large Semantic Knowledge Graphs have SPARQL queries that they run as part of regression testing that search for anti-patterns as well as errors. There are some examples of SPARQL queries that check for anti-patterns in the book's GitHub site in the directory: https://github.com/mdebellis/SemanticKG-Design/tree/main/tests.

OOPS! is a useful tool; however, the question of what an ontology anti-pattern is often depends on the context. For example, OOPS! will identify any property without a defined domain or range as an anti-pattern. Often it is. However, there can be good reasons for leaving the domain and/or range of a property undefined. Reusable vocabularies are one such case; we may want to define properties without a domain and/or range as they will be defined by the ontology that uses the vocabulary. OOPS! is useful, and I encourage people to use it. However, don't blindly accept every OOPS! suggestion. Evaluate the report with the best experts in ontology design on your team before applying its recommended fixes.

Another tool that is useful for anti-pattern analysis is ROBOT from the OBO Foundry. It will be discussed in detail in **Chapter 8**.

Large Language Model (LLM)

There is healthy skepticism surrounding large language models. While concerns about hallucination are real (and we'll be examining the RAG architecture which deals with hallucinations in later chapters), even if LLMs made no progress from where they are today, they provide features that I never expected in a software tool in my lifetime. I'm not going to list all the ways you can use LLMs here. It would make this chapter twice as long and I think it makes more sense to discuss LLMs in the context of specific parts of the life cycle below. Also, I will discuss how LLMs are the perfect complement to Semantic Knowledge Graphs using

architectures such as retrieval-augmented generation and how they revolutionize the design of common data applications and architectures such as Data Catalogs and Data Mesh. Stay tuned.

Summary

- *Developing a Semantic Knowledge Graph is about much more than designing an ontology*. Designing the ontology is the intellectually rewarding part. Making it work in a production environment is usually the hard part. Tasks such as writing code, uploading data, and testing require additional tools.
- *Ontology modeling* is supported by several commercial and open-source tools, but one of the best that anyone working with Semantic Knowledge Graphs should understand is the free, open-source Protégé ontology editor from Stanford.
- *Vocabularies are ontologies that are not designed for a specific system but rather are reusable assets that solve common technical and industry problems.* They can be divided up into three categories: upper models, technical (horizontal) vocabularies, and industry (vertical) vocabularies. Upper models should be lean because they are meant to be the foundation for all ontologies in an organization. An excellent upper model is the Simplified Upper Level Ontology (SULO) Dumontier et al., 2025). For business ontologies, the gist upper model from Semantic Arts is an excellent foundation (Blackwood, 2020). For technical vocabularies, always begin with the ontologies from the W3C (W3C, 2026). For industry and domain ontologies, look for ontologies supported by respected organizations such as SNOMED for healthcare and FIBO for financial services.
- *Semantic Knowledge Graph databases, also known as triplestores, allow you to save Semantic Knowledge Graphs in their native format.* Most of the vendors of triplestore databases provide excellent free versions that allow you to do research and develop prototypes to help you build skills before tackling large deployed systems.
- *While Protégé has SPARQL and SHACL plugins, they are only for education and are not powerful enough to support real development.* Triplestore vendors all have excellent implementations of SPARQL and SHACL as do most RDF and OWL libraries such as RDFLib in Python.
- *GitHub is the tool that is used by most Semantic Knowledge Graph developers to store projects that include ontologies as well as code and other artifacts.*
- *Tools for discovering anti-patterns and performing common ontology refactoring are very useful*. OOPS! is a good tool for discovering anti-patterns (Poveda-Villalón et al., 2014). ROBOT from the OBO Foundry is useful for automated refactoring (OBO Foundry, 2026).

Footnotes

1 https://protege.stanford.edu/support.php

2 https://github.com/mdebellis/SemanticKG-Design/wiki/Exercises

3 https://github.com/protegeproject/cellfie-plugin

4 Note: the raw file path looks different if you are a registered user of the GitHub repository. This is to distinguish raw files that can only be loaded from those that can be edited.

5 https://github.com/mdebellis/SPARQL_Tools

6 https://oops.linkeddata.es/

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_6

6. Requirements Acquisition

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

In this chapter, we'll discuss various techniques and best practices to define requirements for systems that utilize Semantic Knowledge Graphs.

The Role of a System Narrative in Requirements Definition

When defining requirements, it is often useful to begin with a narrative: a high-level description of the goals and functions of the system, expressed in everyday language. The narrative does not list individual features or constraints. Instead, it describes the big picture of the system: what it is intended to accomplish, how it will be used, and what value it will deliver.

A narrative can be especially useful for Semantic Knowledge Graph projects which are often embedded in enterprise-wide data initiatives such as data lakes and catalogs. These efforts are ambitious in scope: no team can deliver the whole vision in one step. A well-crafted narrative both defines the big picture and clarifies that the first phase is just the first step in a long journey.

It creates alignment, sets expectations, and establishes a shared vocabulary. By framing the system's purpose in terms of strategic goals,

the narrative helps everyone from business sponsors to technical teams see how their work contributes to the bigger picture.

Different methodologies have different names for this concept. In Agile, it most closely corresponds to a product vision or, in some cases, large-scale epics. In traditional software engineering, it maps to a concept of operations (ConOps) or system overview. Whatever the label, the underlying idea is the same: start with a clear, high-level story before defining specific requirements.

Example Narrative (Enterprise Data Catalog Project)

> *Our enterprise has data assets spread across dozens of systems: transactional databases, cloud storage, spreadsheets, and legacy applications. Today, employees waste time searching for the right data or recreating data they cannot find. Our goal is to build a semantic data catalog that provides a unified view of these assets, making them discoverable and understandable. The catalog will support key business initiatives such as regulatory compliance and advanced analytics, while also reducing duplication of effort. In the first phase, we will focus on a limited but valuable subset: customer and product data. This pilot will prove the approach, deliver immediate benefits to sales and compliance teams, and lay the foundation for extending the catalog to other domains in later iterations.*

Key Elements of a Narrative

Looking at this example, you can see that its structure essentially consists of answers to the following checklist of questions:

- What is the current situation?
- What undesirable aspects of that situation would you like to change?
- What do you propose to do in terms of new and different capabilities provided?
- What aspects of your operation will those capabilities support, and how will they make things better?

- Where are you going to start?
- How are you going to expand from there?

Epics in Agile

A more narrowly scoped narrative is called an Epic in Agile. In Agile, we define specific requirements with stories, which will be described below. An Epic is a broad description of requirements that is broken down into several stories. It sits at the boundary between narrative and requirement. It traces directly back to the larger enterprise vision, while at the same time specifying a tangible unit of value that can be delivered in an iteration. Here is an example:

> *As a compliance officer, I want a searchable catalog of customer data sources so that I can quickly verify where sensitive information is stored and ensure it meets regulatory requirements.*

Modeling Individual Requirements

From this broader vision, we define specific requirements. In practice, there are three ways to define requirements for a Semantic Knowledge Graph system:

1. *Stories*: Short, role-based descriptions of functionality, widely used in Agile. Stories often use the format: "A <*type of user*> needs to <*perform some task*> in order to <*achieve some goal*>." This isn't always a good fit, but it provides a simple structure that works with many requirements and helps requirements analysts focus on users, tasks, and goals.
2. *Use Cases*: More structured scenarios that describe system interactions step by step.
3. *Competency Questions*: Natural-language questions the system should be able to answer, most often used in ontology design.

Another important distinction among requirements is the difference between functional and nonfunctional requirements. This is one of those tech names that makes you furrow your brow the first time you hear it. Why would we want *nonfunctional* requirements? They sound like requirements that don't work. Of course, the actual meaning is that nonfunctional requirements apply to aspects of the system that aren't specific functions it must perform like processing a payment or listing the most popular videos on a site. Instead, they describe performance characteristics: things related to how many, how fast, how secure, etc. Things like how many concurrent users, how many triples, required response time, and how many transactions per second. These don't fit as well in the formats described above, but they are critical to understand and can make a project that works perfectly during testing a failure when deployed. Even if every feature works perfectly, if the system crashes when deployed with the actual number of concurrent real users, it's a disaster.

Stories

Stories typically follow the format: A <*type of user*> needs to <*perform some task*> in order to <*achieve some goal*>. This provides a simple template to guide requirements documentation. For example:

- As a data engineer, I want to trace the upstream dependencies of a data product, in order to detect how changes in source data will impact downstream analytics pipelines.
- As a data governance specialist, I want to see the full lineage of a dataset, including how it was generated and transformed, in order to verify compliance, audit data usage, and enforce governance policies.
- As a scientist working on climate modeling, I want to be able to find other researchers with specific skills that I need to solve complex problems.
- As a data scientist, I want to query the knowledge graph to discover related datasets and their relationships, in order to build better models using well-understood and properly contextualized data.

This makes stories easy to write, read, and connect to business value. They are best used when you need to capture high-level functional goals that can later be broken down into more technical tasks. The goal of a story is to define one specific business function that the system must perform.

Story Size and Modeling Scope

Here are some heuristics for story size and scope. These same heuristics apply if your team prefers competency questions or use cases. Agility depends on working in increments small enough to understand, test, measure, and deliver within a single iteration. A modeling story (the Semantic Knowledge Graph version of a user story) should represent a coherent but limited piece of work, something that can be designed, implemented, and validated within one sprint. A sprint is typically a two-week iteration.

If a story can't be completed in a single iteration, it should be divided into smaller parts, each of which can yield visible progress and feedback. A good rule of thumb is that most modeling stories should take no more than a few days of focused effort. This keeps the work concrete and encourages frequent validation through reasoning, SHACL, and data tests.

Stories should follow the ***INVEST*** principle:

- *Independent*: Can be developed without waiting for other stories
- *Negotiable*: Open to discussion and refinement
- *Valuable*: Delivers a tangible benefit to users or the data ecosystem
- *Estimable*: Small and clear enough to size or schedule
- *Small*: Fits comfortably within a single modeling cycle
- *Testable*: Can be verified logically or empirically once completed

Keeping stories small supports iteration, adaptability, and continuous validation. The same qualities that make Agile effective in software development are equally essential in ontology and knowledge-graph design and development.

Users of Semantic Knowledge Graphs

In most software development projects, "users" are customers or end-users who interact directly with the system but don't play a direct role in developing it. In ontology and knowledge-graph development, the situation is often different. Semantic Knowledge Graphs are often built for analytic rather than transactional applications. They are often used in systems such as data lakes, data catalogs, and knowledge management rather than online transaction processing (OLTP). Thus, users can often be data professionals: data analysts, data scientists, compliance officers, domain experts, or engineers who build applications that use the graph as a semantic foundation.

In this context, these technical and analytic roles are the users. They are the people who depend on the graph to interpret, integrate, and reason over data. Writing user stories from their perspective keeps the focus on delivering value: better queries, cleaner semantics, consistent terminology, and reliable reasoning.

This concept of technical users as customers is entirely consistent with modern microservice architectures, where the users of one component are often other business units as well as customers. We'll explore these ideas in more detail when we discuss data products in **Chapters 11 and 13**.

Example: Breaking Down an Oversized Modeling Story

Too large (initial story): *As a data engineer, I want to model the employment domain including employees, organizations, job titles, dates of employment, and compensation, so that the knowledge graph can represent all aspects of workforce relationships in one structure.*

This is a classic "too-big" story. It sounds clear, but in reality, it spans several distinct conceptual challenges:

- Modeling n-ary relations between people and organizations
- Representing temporal information such as start and end dates
- Defining controlled vocabularies for roles and job titles
- Managing sensitive data such as salary or pay grade

Each of these requires different modeling decisions, tests, and stakeholder reviews. Attempting them all in one iteration risks logical errors, unclear requirements, and stalled progress.

Here's how this could be refactored into smaller stories, along with statements of the technical steps entailed that show how each smaller story has a testable implementation:

1. *Story: As a data engineer, I want to represent employment as a relation between a person and an organization so that employment data can be linked consistently across systems.*
   Implementation: Create the `Employment` class (n-ary relation, see ontology patterns in **Chapter 7**) linking `Person` and `Organization`. Validate reasoning to ensure each `Employment` involves exactly one `Person` and one `Organization` by defining the appropriate SHACL shapes.
2. *Story*: *As a business analyst, I want to record the start and end dates of employment so that I can query employment histories and durations*.
   Implementation: Add `start_date` and `end_date` with appropriate datatype constraints. Test reasoning over ongoing employments where no end date is given.
3. *Story: As an HR data steward, I want to standardize job titles so that employment records use consistent terminology across organizations.*
   Implementation: Create a SKOS concept scheme or controlled vocabulary for job titles. Link them to `Employment` via `has_role`.
4. *Story: As a compensation manager, I want to associate pay information with each employment so that salary data can be analyzed and governed separately from other attributes*.
   Implementation: Add optional properties such as `has_salary` or `has_pay_grade`, modeled separately for privacy and

compliance.

Breaking a large story into smaller, user-focused increments allows each piece to be modeled, reasoned over, and validated within a single iteration. It turns an abstract goal (*model the employment domain*) into a sequence of testable outcomes. This is central to our principle of progress through continuous validation by delivering working, reviewable ontology modules.

Use Cases

Use cases are more detailed narratives that describe how an *actor* (person or system) interacts with the system to achieve a goal. A well-formed use case typically includes

- Title (short, action-oriented)
- Primary actor
- Preconditions (what must already be true before the use case starts)
- Main success scenario (the normal flow of interaction)
- Alternate flows/exceptions
- Postconditions (what is true when the use case successfully completes)

Use cases provide more structure than stories and are especially helpful if your team is already working with business analysts or testers who are used to them.

Competency Questions

Competency questions are written in natural language and provide specific examples of questions that the ontology should be able to answer. They help define the scope of the ontology and serve as a check that the modeling effort supports real information needs.

Examples include

- Which scientists have experience with the Hubble telescope?
- What data must adhere to the European Union's General Data Protection Regulation (GDPR)?

- Which region showed the most growth in revenue, and what factors drove that growth?

Good competency questions are

- Clear and precise
- Relevant to the domain
- Answerable from the ontology once complete
- Varied in scope, ranging from simple retrieval to more complex inference

Selecting the Appropriate Requirements Model

Competency questions are the most common approach in knowledge graph projects because they connect directly to query and reasoning capabilities. In many books and papers describing how to build knowledge graphs, it is taken as a given that competency questions are the one and only way to define requirements for an ontology. However, there are good reasons to consider using stories or use cases instead of, or alongside, competency questions, depending on the audience and project context.

One key reason for this is that any effort always fits within a bigger picture. An industry Semantic Knowledge Graph is going to be part of larger systems with requirements that are defined by analysts using stories or use cases. When defining those requirements, we try to think in terms of business functions, not technology. As a result, it's quite likely that use cases or stories developed outside of the knowledge graph team will also be relevant to their work. Insisting on only using competency questions can create unnecessary division between other developers, the business users, and the knowledge graph team.

The main justification I hear from ontologists for competency questions is that they map very cleanly to Description Logic. This is not how a good developer thinks. A good developer tries to accommodate the business users; she doesn't expect the business users to think in ways that make the developer's life easier.

Another issue with competency questions is that they give the impression that the goal of a knowledge graph is to answer questions. That is certainly ***a goal***, but by no means ***the only goal***. The goal of a knowledge graph is to solve business problems as part of a larger system, not just answer questions.

For example, consider an ontology created to support a travel agency. A competency question might be: *List all flights between two cities for a given time period sorted by price.*

Now consider how the same requirement would be defined as a story: *As a customer service representative, I want to list all flights between two cities for a given time period sorted by price in order to give the customer options so the customer can choose the flight that is the best match to their needs.*

The competency question does not explicitly include the critical details about the user (the *customer service representative)* nor the *customer*. It omits that the goal is not to answer a question but to provide information to the customer so that they can make the best decision. The competency question focuses on answering questions. The story focuses on solving business problems and will hence often include additional information that may be omitted from a competency question as in this simple example.

Stories, because of the added context they bring out, are also better for eliciting thoughts about desirable follow-ups. Knowing that a list of flights sorted by price will be used to pick an ideal flight begs different follow-up questions (e.g., about what filtering might be done with the answers) than if, say, the purpose was to prioritize how many limousines vs. cabs to place at the airport to meet each arriving plane.

For this reason, I generally recommend *stories* because they are the simplest: they offer a clear, repeatable template, and they keep attention focused on solving business problems rather than simply answering questions. They are also more likely to be familiar to other developers and users who work with Agile methods.

However, if a team already has a process for writing *use cases* or *competency questions*, it will often be best to simply stick with what the

team is already familiar with. Remember one of the core Agile principles: *working software is the primary measure of progress*. Requirements and other documentation only help us get there. It is far better to spend time writing good competency questions that everyone on the team understands rather than to waste energy debating which requirement format is best.

For example, one of the first, most successful, and most knowledge-rich ontologies is Gene Ontology (Ashburner et al., 2000). Projects such as this and others in the OBO Foundry (Jackson et al., 2021) have doctors and other specialists as full-time team members, and as a result, they are very comfortable with competency questions. I would never encourage a team like that to disrupt its process by forcing the team to change how they model requirements.

Analyzing Requirements Documents

Analyzing the text of requirements documents is a great way to create initial classes, properties, and axioms. Table 6-1 provides heuristics for mapping parts of speech in requirements documents to the most common OWL entity types.

Table 6-1 Heuristics to Map Parts of Speech to Ontology Entities

A technology from Natural Language Processing called Named Entity Recognition (NER) automates the heuristics described in Table 6-1. Existing NLP libraries such as the spaCy Python library (explosion.ai, 2025) provide NER tools to automatically recognize common entities such as people, organizations, and locations in text documents. This technology has existed for quite some time. Recently, various vendors have begun incorporating it into their products. Combined with improved recognition models, the potential is to do far more than recognize instances of a handful of high-level classes but also recognize domain-specific classes with more detailed information. For example, not just recognize that the string “Michael Jordan” refers to a

person but to a basketball star. I'll show an example of such a tool in **Chapter 8**.

Define Metadata

When gathering requirements for a Semantic Knowledge Graph, one of the most important steps is to define what kinds of *metadata* will be needed. Metadata is "data about data" that allows a system to be understandable, governable, and usable over time. In industry practice, metadata is typically divided into semantic, operational, social, and technical categories.

It is worth noting that in the ontology community, the term metadata is usually reserved for things like provenance, cataloging, and descriptive annotations (e.g., Dublin Core, SKOS, PROV-O). Ontologists often do not think of Semantic models as metadata, because the ontology is the model of the domain. However, in the broader data-management world, semantic models are consistently treated as a kind of metadata, called *semantic metadata*. In this book, I use the ontology definition as the default because it is useful to distinguish the kind of metadata that specific vocabularies (e.g., Dublin Core, SKOS, PROV-O) are designed for vs. semantic metadata which is modeled in OWL. When referring to semantic models as metadata, I'll explicitly use the phrase *semantic metadata*.

At the requirements stage, the goal is not to focus on the models, but to determine what kinds of metadata will be needed and for what purposes. In later stages, when we do conceptual and design modeling, we will implement these requirements using OWL axioms, SHACL shapes, and other representations. Agile methods encourage iteration, so requirements and modeling often evolve together, but distinguishing between "deciding what metadata we need" and "authoring the metadata models" keeps the process clearer.

The key categories of metadata relevant to requirements are

- *Semantic Metadata*: Defines the meaning of the business, scientific, or engineering domain. At the requirements stage, we identify which

domain concepts, relationships, and constraints will need to be captured in a computable form. This does not mean writing OWL axioms yet, but stating the need for them (e.g., "the system must represent scientists, their organizations, and their publications, with the ability to infer seniority or collaboration patterns"). Later, OWL ontologies such as FIBO, SNOMED, GoodRelations, or gist can fulfill these needs.

- *Operational Metadata*: Captures how data is created, maintained, and accessed. Requirements in this area specify the need to track provenance, authorship, versioning, and access control. Standards like Dublin Core, BIBO, and PROV-O provide ready-to-use frameworks to satisfy such requirements.
- *Technical Metadata*: Specifies the data types, formats, and integrity rules that enable reliable storage and exchange of data. Requirements may call for validation rules, serialization formats, or mappings to existing databases. SHACL, XML Schema datatypes, and R2RML mappings are typical technical solutions.
- *Social Metadata (sometimes overlapping with operational metadata)*: Emphasizes ownership, rights, and responsibilities. Requirements here ensure that governance and compliance issues are addressed, such as specifying who owns which data and under what conditions it can be shared. In the past, this was often grouped with operational metadata. However, new approaches to modeling data such as Data Products and Data Mesh as well as legislation such as HIPAA in the United States and the General Data Protection Regulation (GDPR) in the European Union require this to be considered a category of its own.

These categories are summarized in Table 6-2.

Table 6-2 The Four Types of Metadata

Low-Fidelity Prototyping

It might seem strange to discuss user interface prototyping in a book about Semantic Knowledge Graphs. However, our focus is the systems that include Semantic Knowledge Graphs, not just the ontologies and knowledge graphs. For eliciting system requirements, I've found UI prototyping to be a very effective tool. Domain experts often have difficulty describing their expert knowledge. However, everyone is good at telling you what's wrong with a design. A user interface prototype is a design format that anyone who has used computers can understand and give feedback on. User interface prototypes can help communicate requirements and planned functionality in a way that is more immersive and understandable than requirements documented only in text.

One technique that I recommend is low-fidelity prototyping. There are powerful UI prototyping tools available that enable a good UI/UX designer to create a prototype with sophisticated graphics and responses to input that mimic what the actual system will do.

There is no question that UI prototyping tools will be needed. However, for many systems, it can be very powerful to simply start with much simpler low-fidelity prototypes that have minimal graphics and that the designer walks through rather than acting as a hypothetical user and clicking on buttons and selecting from menus. Even the lowest fidelity of just starting with piles of blank pages, pencils for everyone, and having developers and business users iterate over possible initial designs can still be useful. One reason is precisely ***because*** current UI prototyping tools are so good. When a user sees a UI mockup with widgets, colors, sophisticated graphics, and interaction, they are much less likely to give feedback that completely ignores the initial design and are more likely to suggest tweaks instead. Thus, starting with a UI prototype in tools like Figma that looks like a finished product risks locking in the UI design too early and not considering enough alternatives.

As with Agile methods, I've seen the dramatic difference that using low-fidelity prototyping can have. In both cases, using such techniques creates a completely different environment compared to older software

development methods. Rather than have users who resist change, you have users who are real members of the team. Not only do you get much more and better input from users, but the entire relation between developers and users changes from reluctant collaborators to a true team with shared values and goals. This creates better systems and makes the chances of rapid adoption and ROI much greater because the system is now being pulled by the users rather than being pushed upon them.

Summary

- *Defining requirements for systems begins with clarity of purpose. A well-crafted system narrative establishes the strategic vision and aligns stakeholders before individual requirements are specified*. From this foundation, teams can express requirements using stories, use cases, or competency questions, each of which has strengths depending on the project context.
- *Regardless of format, requirements should be small, testable, aligned with iterative development principles, and intuitive for the business and domain experts*. Stories are often the most accessible format because they focus on delivering business value and integrate naturally with Agile development practices. Competency questions are especially useful in ontology design, as they connect directly to query and reasoning capabilities. Use cases provide additional structure when detailed interaction flows are needed.
- *Nonfunctional requirements define aspects of the system that don't define specific business functions, but rather the constraints for performance, security, uptime, and usability* that the system must meet. They are as essential to rigorously define as domain models and business functions.
- *Attention must also be given to metadata requirements: semantic, operational, technical, and social.* Metadata determines how the system will be governed, validated, and extended over time. Techniques such as requirements-document analysis and low-fidelity

prototyping can further clarify needs and improve stakeholder engagement.

- *Ultimately, the goal of requirements acquisition is not to produce documentation for its own sake, but to define clear, actionable objectives that guide modeling, implementation, and validation*. In the next chapter, we turn from defining requirements to translating them into conceptual models.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_7

7. Conceptual Modeling

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

> *It is easy to be lured into the thinking that more is better. It's not. Smaller and simpler ontologies are easier to understand and use and more computationally efficient as well. A good principle to use is "when in doubt, leave it out."*
> —Michael Uschold Demystifying OWL for the Enterprise

Conceptual models are the most abstract type of model. They are also called analysis models. The goal of a conceptual model is to define the business problems in terms that don't make technological commitments.

Many teams begin conceptual modeling with simple visualization tools such as mind maps. Mind maps and similar tools should be treated as provisional sketches that spark discussion. These tools are the equivalent of sketching on a whiteboard: they make it easy to brainstorm, cluster related ideas, and communicate quickly with nontechnical stakeholders. They can be useful for getting people engaged before moving into formal modeling. However, the same informality that makes them approachable also can make them problematic. The lines and boxes lack explicit semantics. Does a connection mean part of, is a, causes, or something else? The intended meaning exists in the mind of the person who drew the diagram. Different people who see the same diagram bring their own unstated context, and as a result, the team may think they've reached a consensus only to find they really haven't when the informal model is turned into OWL. As a result, such diagrams can slow up progress because they often require several rounds of iteration to go from boxes and arrows to classes, subclasses, properties, and axioms.

Before we go further, it is worth noting the minimum concepts that every conceptual model should include. At this level, you need only a few basics: *classes* (the categories of things in your domain), *individuals* (specific examples that belong to those classes), and *properties* (relationships and attributes that connect classes and individuals). Properties divide into *object properties* (linking individuals to other individuals), *data properties* (linking individuals to literal values such as numbers or strings), and *annotation properties* (properties that can be asserted on any entity, including classes and other properties).

Finally, remember that both classes and properties can form hierarchies, so you can capture simple taxonomies without yet worrying about complex logical constraints. With just these elements, you can build a conceptual model that is both rigorous and easy to understand.

As much as possible, every member of the modeling team should understand the basic concepts of OWL. An OWL model provides the high-level abstraction required for a conceptual model as well as the rigor of set theory and logic. Tools that provide visualization of OWL models provide the same intuitive graphic views as informal models but with rigor that such tools lack. In addition, using OWL to define conceptual models allows us to reuse vocabularies such as SKOS, PROV-O, and Dublin Core that reflect designs created by some of the best experts on how to model technical domains such as metadata and industry domains such as retail and healthcare.

One of the most important heuristics for designing a conceptual model is expressed by the quote from Michael Uschold: *when in doubt, leave it out*. Keep the number of terms to a minimum.

A particularly effective tool for early conceptual modeling is WebProtégé (Tudorache, Tania, et al., 2020). Unlike the desktop version of Protégé, WebProtégé does not support a reasoner and therefore does not allow defining axioms. This makes it well suited for creating simple, high-level ontologies that capture agreement on the core concepts and relationships in a domain. Because it is hosted, WebProtégé also supports collaborative editing in real time, allowing teams to work together, even those who are not experts in OWL, to shape and build the emerging model. WebProtégé functions as both a consensus-building tool and a lightweight entry point into ontology design.

When using WebProtégé, it is a best practice to have a technical expert review any ontologies that are created or loaded. See **Chapter 12** WebProtégé in DeBellis (2021). The reason is that WebProtégé can do some unintuitive things to IRIs. For example, if you load an ontology into WebProtégé, you would expect that the ontology IRI is the default to use for new entities, but that isn't the case. I often use WebProtégé to build a consensus on a model. When I do, I always make sure there is a technical person responsible for setting up the initial model so as to avoid the minor problems that WebProtégé can cause. These minor problems are easy to fix if you understand the basic concepts of IRIs that we've covered. However, if you don't address them early, they can impose a large technical debt that becomes more and more costly to repair.

Design Principles for Conceptual Models

When creating conceptual models, it helps to keep in mind a few general principles. These are adapted from Tom Gruber's classic paper on ontology design (Gruber, 1993):

- *Clarity*: Define concepts as clearly as possible, using ordinary language and examples. The goal is that someone outside the project can understand what each concept means, without requiring insider knowledge.
- *Coherence*: Make sure the model "hangs together." If you add a new concept or relation, it should not contradict the assumptions already present. At the conceptual level, this means checking for consistency in the way the model represents your domain.
- *Extendibility*: Avoid painting yourself into a corner. A good conceptual model can be refined or extended later without having to be discarded. For example, start with "Employee works for Agent" rather than "Employee works for Corporation," unless you're certain that employers in your ontology will always be corporations. That way you can also model working for an individual, a university, and being self-employed.
- *Minimal Commitment*: Commit only to the distinctions you really need at this stage. Leave room for future refinements. If you're not sure whether something should be modeled as a class, property, or tag, note the idea but don't over-specify. The model should be just detailed enough to support the current conversation.
- *Minimal Encoding Bias*: Don't let the quirks of whatever drawing or modeling tool you happen to use drive the structure of the model. The conceptual model should reflect the domain, not the tool. A common form of encoding bias today arises when modelers unconsciously force OWL ontologies to look like UML class diagrams. For example, a tool may visually encourage attaching properties and attributes directly to classes, leading the modeler to treat all properties as if they were simple fields rather than first-class relationships. This can result in missed opportunities to model rich semantics such as property hierarchies, inverse properties, or restrictions that are central to OWL but awkward to express in UML-style diagrams. The ontology ends up reflecting the drawing notation rather than the structure of the domain.
- *Create Instances*: This is the most common error I see when new users design ontologies. They focus on just defining the TBox (classes, properties, rules, and axioms) and don't create any instances. Unfortunately, many of the errors that the reasoner can detect in your model are only visible when you have representative instances. In addition, it is easy to think your model is complete if you have no instances, only to discover that you need to refactor or add to the ontology once you start loading

instances. Remember, it is possible to do design bottom up as well as top down. That is, start with some instances and think about the data that needs to be stored and the relations between those instances. LLMs can be very useful to define instances. For example, most of the instances and almost all of the property values for the People Ontology were provided for me by ChatGPT. I began with a few people I admire such as Einstein, Turing, and F. Scott Fitzgerald and then asked ChatGPT to provide data to make the ontology less biased toward Western men such as the names of famous people in Asia, names of famous women scientists, and property values such as birthdates, employers, and family relations. The best way to do this is to ask the LLM to generate an INSERT DATA SPARQL update. If you ask the LLM to generate a new version of the ontology, you will often run into memory problems where the LLM can't keep all the existing data in the ontology in its memory while it also adds more.

Conceptual Modeling and Design Modeling

The fundamental difference between conceptual and design modeling is that in conceptual modeling, we focus on understanding the problem as clearly and unambiguously as possible. In design modeling, we consider issues such as size of the model, number of instances and triples, required response time, and security constraints. In addition, in design modeling, we focus on all the bad data that is unfortunately almost always something we have to deal with. As a result, the way we create these models has different priorities.

In conceptual modeling, we

- Create inverses for most object properties; this can be incredibly useful when editing early versions of an ontology in Protégé because it doubles the chance that you will see incorrect or missing relations.
- Use transitive and symmetric properties where appropriate.
- Add any useful axioms.
- Define domains and ranges in OWL.
- Use OWL rather than RDF or SHACL.
- Use SWRL to understand and implement business rules.
- Use superproperties to organize data properties.

In design modeling, we

- Consider if inverses are really needed at run time rather than being used to understand the emerging conceptual model.
- Consider the cost (many additional triples) of using transitive and symmetric properties.
- Analyze the performance cost of using axioms. Consider replacing axioms with SHACL rules and/or SPARQL queries (to find problems) and updates (to fix problems).
- Evaluate the cost of using range in OWL data properties, since real-world data may often cause ontologies to be inconsistent. Transform OWL ranges to SHACL constraints.
- Consider using `rdfs:Class` and SHACL rather than OWL for greater efficiency.
- Implement SWRL rules in SPARQL, code, or more efficient rule engines.
- Avoid superproperties for data properties because they create redundant triples.

Buckets

One of the most common things we do with a model is find related things (concepts, people, products, etc.) and group them together. A strength of OWL is that there are several ways to do this. We can make related things instances of the same class, we can make them part of a collection, or we can create *tags* to annotate an instance to show that it belongs to a specific category. These different modeling techniques are sometimes called *buckets* (Uschold, 2018) because they are all different ways to group things together, i.e., put them in the same conceptual bucket. However, it can sometimes be a challenge to decide

which bucket to use. This section describes the three types of buckets and how to decide which one to use to create a new bucket for things that are related. The three types of buckets are

1. *Instances*: In the People Ontology, each Person is an instance of the class Person. This is the same as saying there is a set called Person, and each individual Person is a member of that set. In addition, there are subclasses of Person such as Child, Female, and Employee. These are smaller buckets than Person because anyone that is an instance of one of the subclasses is also an instance of Person. Instances are the right choice when the members of a bucket share many properties and behaviors. For example, all people have a birth date, an age, a mother, a father, etc.
2. *Collections/Roles*: A collection/role bucket is used when instances are grouped through a property that relates them to a common grouping. For example, in the People Ontology, the property `is_employed_by` links a `Person` to some `Agent`. The `Agent` is typically an `Organization` such as a `University` or `Corporation`, but one `Person` can also be employed by another `Person`, so we use the more general class `Agent` as the range. Collections are especially useful when modeling membership, components, or containment hierarchies (e.g., a Bill of Materials in manufacturing, members of the Union of Concerned Scientists, the containment relations between nations, states, and cities). A Protégé plugin to view these kinds of component and subcomponent hierarchies is the **Individual Hierarchy Tab**. Figure 7-1 shows a simple example: a component hierarchy for two computers in the People Ontology shown in the **Individual Hierarchy Tab**. It also shows the property assertions for one of the computers, as well as how the transitive superproperty allows us to see all the components.

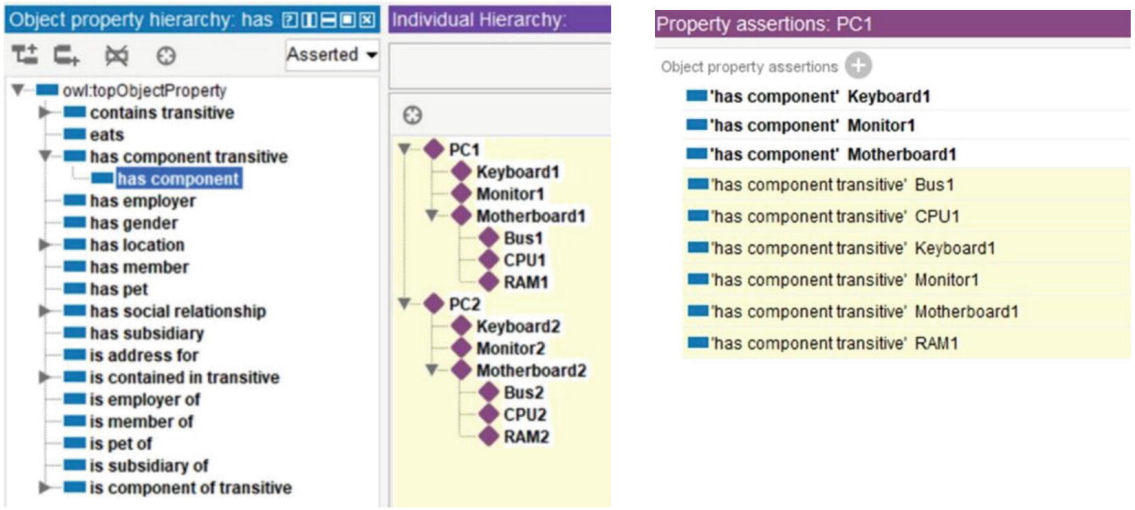


Figure 7-1 Individual hierarchy tab and property assertion view in Protégé

3. *Tags*: Tags are a lightweight way to group instances by assigning them a categorical label. This is typically modeled as an enumerated class, where we explicitly define all the possible values. For example, the `Gender` class is defined as the set `{Female, Male, Non_Binary}`. A `Person` can then be linked to one of these tags using the property `has_gender`. Unlike instances or collections, tags do not usually imply that the tagged individuals share many properties or form part of a structure. They simply categorize an individual along some dimension, often one that is orthogonal to class or collection membership. Tags are most appropriate when you want to model "X has category Y" (e.g., "Pizza has spiciness-level Hot"), rather than "X is a Y" or "X is part of Y."

   One way to decide which kind of bucket to use is to pay attention to how we talk about the relationship in natural language:

- If we say "X is a Y," that points to the *instance bucket* (e.g., Albert is a Person).
- If we say "X is part of Y" or "X belongs to Y," that suggests a *collection bucket* (e.g., "John is employed by IBM," "San Francisco is contained in California").
- If we say "X has category Y," that's a good fit for a *tag bucket* (e.g., "Albert has gender male").

Some additional heuristics:

- If your bucket is commonly used as the domain for properties, that is a good indication it should be a *class*.
- If your bucket is part of a *folksonomy*, i.e., annotations that can be created on the fly by people to add context to an instance, that is a good sign that it is a *tag* bucket. See **Chapter 13** for more on folksonomies.

Tags can be modeled as instances of a class (and hence, values of an object property) or as strings and values of a data property. At first, it may seem tempting to use strings. They have slightly less overhead, and if you want to create new tags, you can just write new string values. However, instances and object properties have many features that often make them preferable to strings for tags. The benefits of using individuals rather than strings to model tags are shown in the following exercise.

EXERCISE 7-1: DEFINING TAG VALUES

To understand why instances are often better than strings for tags, bring up the People Ontology in Protégé and navigate to the **Individuals by class** tab. Expand `Agent` in the **Class hierarchy** view (upper left corner) and click on `Person`. In the **Individuals** view right below, you should see "`Alan Turing`" as the first instance of Person. You can also use the search icon in the upper right corner (the magnifying glass) to search for him. Click on Turing and you will see the values asserted about him in the **Property assertions** view (lower right corner).

Click on the + next to **Object property assertions** in that view. This brings up the user interface to add a value to an object property. Type: 'has p and hit <control><space>. Make sure to include the quote symbol and make sure not to have an underscore between "has" and "p". Recall the distinction between completion for the IRI vs. the label. Since most of the entities you deal with have labels, most of the time, you should start with that single quote character. Since there is more than one object property that starts with the label "has p," you should get a menu of possible completions. Select the property "`has profession role`." Now, for the value (the field to the right), enter 'Cmp and again <control><space>. Nothing completes. Try just adding the value as it is. You can't. Protégé knows that the value must be an instance, and you aren't giving it the name of one. If you enter 'Com and do <control><space>, that should give you the value "Computer Scientist" which you can enter.

Now go to the **Data property assertions** for Turing in the same view right below. Click on the + next to **Data property assertions**. The UI is a bit different because you have a tree with all the data properties under **top_Data_Property**. Navigate to the data property `profession_string`, select it, and type "Cmp" (don't add the double quotes), and hit <control><space>. Nothing happens, as expected. Try just entering "Cmp". You can. What you've just experienced illustrates why instances can be better than strings for tags. With instances, you have a well-defined set of allowable values. If you try to enter a tag that isn't an entity, you can't do it. If you model tags as strings, you can type any arbitrary thing you want as a tag. In fact, Protégé doesn't exploit this as much as it could. It only checks for instances, but it could do more. When you create your own user interfaces, you can make sure that not only must values for tag properties be instances but that they are instances of the appropriate class. You can also make it easy for people who have the authority to easily add new tags and make sure they are added to the knowledge graph and immediately made available to other users.

This is not to say that tags should always be implemented as instances. There are also good reasons to model them as strings:

1. Easier and faster to add, rename, and delete tags modeled as strings.
2. The majority of current systems utilize string tags rather than OWL individuals.

At the conceptual level, we can also combine buckets. For example, in the People Ontology, we model gender as a tag, but we also have defined classes such as `Man: Adult and ('has gender' value Male)` and `Girl: Child and ('has gender' value Female)`.

Multiple Inheritance

Multiple inheritance is when a class can have more than one direct superclass. For example, a `Teaching_Assistant` can be modeled as a subclass of `Graduate_Student` and `Employee`. Some methodologies strongly discourage the use of multiple inheritance. There can be issues with it. One such issue is that if parts of your model are ultimately going to be implemented in an OOP language, many OOP languages (e.g., Smalltalk) don't support multiple inheritance and even for those that do, many OOP developers discourage the use of multiple inheritance. However, these are not conceptual issues; they are issues for the design and implementation models. Using multiple inheritance can be a powerful way to model at the conceptual level although it may end up that as we move downstream, we change multiple inheritance to a role. See the n-ary properties ontology pattern for more on roles. Transforming from multiple inheritance to an n-ary relation modeled as a class is a common refactoring pattern.

Basic Modeling Decisions

In this section, I'll describe some of the most common modeling decisions and guidelines for how to make the correct decision.

Modeling the World vs. Modeling a Domain

A common source of confusion in ontology design is treating the model as a description of "everything that exists" rather than a precise representation of what matters in a specific domain.

For example, in everyday language, people, organizations, data, and software might all be called "assets." In a data catalog domain, however, only certain things are considered assets: they have versions, lifecycles, implementations, and governance rules. People and organizations act on assets; they are not assets in this sense, although in an HR domain they may be.

Another good example is one I recall from my first college physics class. In common language, "velocity" and "speed" are synonyms. However, in the physics domain, they have more specific and rigorous definitions. Velocity is a vector; it models how fast something is moving (distance divided by time) as well as the direction in a plane or three-dimensional space. Speed, on the other hand, is a scalar. It only measures distance over time rather than the direction.

This is one of the most common issues I've seen that cause conceptual modeling to take far longer than necessary. Effective conceptual models reflect the semantics of the domain, not the ambiguities of natural language. The goal of modeling is to create rigorous, objective definitions with unambiguous boundaries. That is, our goal is akin to scientific models which are the exact opposite of natural language. Natural language is inherently ambiguous. Natural language semantics differ according to the experiences and context of each individual. This is one of the few things that linguists as different as Chomsky (Chomsky, 2016) and Lakoff (Lakoff, 1987) agree on.

Class or Instance?

At first, this seems very straightforward. However, one of the most common mistakes when people develop their first ontology is not knowing when to shift from modeling subclasses to instances. I think one of the reasons (which we'll explore more in **Chapter 12**) is that thinking in terms of trees that go

from abstract concepts to more specific ones is very natural. As a result, when people first start using Protégé and they cross that boundary between classes (i.e., sets: Animal, Dog, Hedgehog) to instances (elements in the sets: Fido, Dinsdale), they often keep going and add the instances as subclasses.

The core guideline is simple and in most cases unambiguous: are we talking about a "kind of" relation or an "is a" relation? If the former (e.g., rottweiler is a kind of dog), then it is a subclass relation. If the latter (e.g., Fido is a rottweiler), then it's an instance relation.

However, it isn't always that straightforward. One example is a product catalog. You have products such as a tablet computer. At first, it seems that categories such as `Basic_Tablet`, `School_Tablet`, `and Pro_Tablet` would be classes. However, consider two different scenarios:

1. *Scenario 1:* A vendor that is focused on selling less expensive copies of the iPad. One of the ways they keep down costs is by not providing alternatives so that each customer can make their own customization. That is, if you get their basic tablet, there is just one version with a certain color, flash memory, screen size, etc. Also, their online catalog is only informational. They don't have the infrastructure to support online purchases. People look on the web to understand the options then go to their store to make their purchase.
2. *Scenario 2:* Apple's website provides basic models and standard features, but each customer can customize their iPad with more memory, a different color, etc. Their website not only allows you to purchase online but customize and purchase your product.

In the first scenario, the concepts `Basic_Tablet`, `School_Tablet`, and `Pro_Tablet` should be instances, not classes. Since the tablets can't be changed and since the customer can't create an order online, there is no need to have instances of the kinds of tablets. The basic kinds of tablets are the instances in this case. In the second scenario, we need to create a new instance for each order, to track the specific customer decisions and track the object as it is sent from Apple to the customer. In that case, the concepts `Basic_Tablet`, `School_Tablet`, and `Pro_Tablet` are classes, and the individuals are the specific instantiations of those classes with the relevant customizations for each customer and additional information to facilitate purchase and shipping of the customized instance.

The point is that it isn't always straightforward to determine the boundary between classes and instances. If the "kind of" or "is a" heuristics don't give you an obvious result, the way to decide is based on how the data will be used. If you need to track instances of the class, to assign them state by asserting property values, then you want to model them as instances. If you are describing categories or types of objects, then you should be using classes.

Also, this example illustrates that while the linguistic heuristics in Table 6-1 provide good guidelines, they are just heuristics and shouldn't be applied without also thinking about how the information will be used.

What Type of Property?

Recall that there are three types of properties: *object properties*, *datatype properties*, and *annotation properties*. Here, the decisions are pretty straightforward. If you want to model metadata about some entity in the ontology such as its label, who created it, when it was created, etc., that would be an annotation property because metadata applies to all the entities in the ontology; however, only annotation properties can be defined for classes and properties. If the range of your property is a simple datatype such as a string or an integer and it only needs to be applied to instances, then you want a datatype property. If the range of your property is a structured object such as an address or person, then you want to use an object property. However, address is another example where the modeling choice isn't always straightforward. It is common for systems outside the knowledge graph to represent an address as a large string; again, it all depends on the context of your specific domain.

A Note on Two Kinds of Metadata

The term metadata is used in two related but different ways in Semantic Knowledge Graphs, and this distinction is essential for understanding property types in OWL.

First, there is *metadata about the ontology itself*. For example, labels, definitions, authorship, versioning, and editorial notes. This kind of metadata describes the vocabulary and model, not the real-world entities being modeled. In OWL, such metadata is usually expressed using annotation properties, because annotation properties are the only properties that can be applied uniformly to classes and properties, as well as individuals.

Second, there is *metadata about data assets modeled by the ontology*. For example, who created a dataset, when a record was generated, the status of a data product, or its quality metrics. This information is also "metadata." However, it is still part of the domain being modeled. As a result, this type of metadata can be modeled by any ontology entity. We have classes that describe different types of this kind of metadata; we represent this metadata on object, data, and also annotation properties. We'll see examples of this in the Data Catalog example in **Chapter 13**.

Directional Object Properties

We will discuss Domain-Driven Design (DDD) more in **Chapter 13**; however, one concept that is worth mentioning from DDD that is relevant to the conceptual model is the concept of establishing a *direction* for associations. Associations are what UML calls object properties, links between two different objects. I'll just use the term object properties to avoid confusion, but for the purposes of this discussion, they are synonyms. Evans (Evans, 2003) points out the importance of establishing a direction for object properties. Of course, there will often be properties that have many-to-many relations where there isn't really an obvious direction. However, for many properties, such as `has_prime_minister` (with domain `Nation` and range `Person`), it is important in DDD to define the direction. We are more likely to ask who is the prime minister of Canada than we are to ask of a Person in the database which nation they are prime minister of.

This is even more relevant when doing conceptual modeling in OWL because it is often useful in the early stages of the conceptual model to always define an inverse. Doing this will help highlight errors in the model. For example, if nations are being defined as the value of `has_prime_minister` when they shouldn't, we double our chances to discover this when we have an inverse such as `is_prime_minister_of` defined. Once we have a good understanding of the conceptual model as well as of the amount of sample data, we can decide if it is worth keeping the inverse property. Defining the primary direction of the property and its inverse makes it trivial to remove the inverse if we decide to do so in the design phase. When you have two properties that are inverses, the one that is the starting point for the direction (i.e., the one that will not be deleted later if you decide you don't need the inverse) should be

- The property whose IRI usually uses the *has_property* naming standard. For example, in the People Ontology, the direction for `has_pet` is from a `Person` to their pet.
- The inverse property usually follows the *is_property_of* standard. I say usually because there are circumstances where it can be more intuitive to not use the *has_property* and *is_property_of* naming standards. For this example, we define the inverse as `is_pet_of`.
- The domain and range are defined on the primary property. In this example, the domain and range are defined on `has_pet`, and the reasoner sets the appropriate domain and range for the inverse.
- Values are associated on the primary property and then inferred to the inverse. In this example, we define instances of the `Animal` class as the value for `has_pet` on some Person and the reasoner infers the `Person` instance value for `is_pet_of`.
- Following this standard makes it trivial to delete the inverse if we decide in the design phase that we don't need the inverse. We can just delete the inverse and all other definitions and values remain.

Subclass or Collection/Role?

One of the hardest modeling decisions is whether a bucket should be represented as a subclass or as a role. A common example is that a person is typically modeled as a class while the type of work (developer, architect, manager) for a project team member is typically a role. Uschold's bucket concept usefully forces this question into the open: are we talking about a kind of thing (a subclass) or a way an instance participates in some context (a role)? If we are describing buckets that tend to remain static, then we typically use a class. If we are talking about buckets that vary, then we typically use a role. A person is always going to be a person, but their role on a project can be quite fluid, both from one project to another as well as even over the lifetime of the project.

Another good heuristic is if removing the bucket membership would imply changing what the individual is, then the bucket should be a class. For example, mammal, corporation, university, and journal article are typically classes because they identify the kind of thing something is and seldom change.

This heuristic also means that when the bucket reflects context-dependent situations or membership, the bucket is likely a role. A role represents a temporary, relational, or situational status. For example, employee, student, and customer are typical roles. The collection/role bucket is best understood in terms of function, responsibility, or state rather than intrinsic nature.

A related modeling decision is whether to model a role as a class or a tag. If the value of the role is just an identifier and doesn't need to include additional information, then it is best modeled as a tag (in OWL this is an enumerated class like the `Gender` and `Role` classes in the People Ontology). For example, if all we need to know about a team member is what role(s) they have on the project (developer, manager, data scientist), then we just use a tag. If, on the other hand, we want to have additional information about their role such as what skills are required and whether they have they had this role on previous projects, then we model the role as a class. That allows us to create an instance of the role for each person that can be customized with data that describes the person and the role.

Semantics or Efficiency?

Finally, a trade-off that is inevitable with a high-level language such as OWL is how much semantics you want vs. how efficient the model needs to be. For example, defining an inverse or making a property transitive will at a minimum double the number of triples for that property. Every axiom you add will make the reasoner slower. The best guideline here is based on what stage you are at with your ontology. When creating a new ontology, it can be very useful to define inverses for all object properties following the `has_property` and `is_property_of` naming convention. Having inverses will surface potential problems more rapidly because you double the chance that you will see a problem when you have an inverse for a property. In the early stages of building your conceptual model, it is best to focus on understanding the business problem. That is, take advantage of every tool that OWL offers for defining your model. When you understand the domain and have an ontology that has no reasoning errors and has been analyzed by tools such as ROBOT and OOPS! for anti-patterns, that is the time to move from conceptual modeling (where you work with representative samples of data) to design modeling (where you work with large data sets of the size you'll encounter when your system is deployed) and to consider reducing/refactoring some entities or axioms into models that provide less information but are more efficient.

Ontology Reuse

One of the great strengths of the Semantic Web is that it allows us to take advantage of existing reusable ontologies called vocabularies. There is an abundance of well-established vocabularies developed by experts in knowledge representation, data science, and specific application domains that we can build on. Reusing these ontologies provides multiple benefits: it saves us time by avoiding reinventing the wheel, it aligns our work with best practices, it makes our ontologies easier for others to understand and integrate, and most importantly, it allows us to focus on the core competencies that differentiate us from

the competition. By standing on the shoulders of ontologies defined by experts, we can direct our energy to modeling the concepts and relationships that truly differentiate our domain and expertise.

There are three broad categories of reusable vocabularies:

1. Technical or horizontal vocabularies
2. Domain or vertical vocabularies
3. Upper models

Technical Vocabularies

For technical vocabularies, there are a few groups that have vocabularies that are virtually universally utilized for specific topics. If there is a well-established technical standard, you should use it unless you have a strong reason not to. Doing so improves interoperability, reduces redundancy, and lowers the cost of long-term maintenance. Some of the most important organizations that define technical vocabularies are

- *Dublin Core (DCMI)*: The de facto standard for basic metadata such as who created an entity. We'll discuss this in more detail below.
- *W3C*: W3C publishes most of the core technical standards such as SKOS (Simple Knowledge Organization System), PROV-O (Data Provenance), DCAT (Data Catalog), and Time.
- *ISO*: Provides foundational standards such as ISO 8601 (dates), ISO 639 (languages), and ISO/IEC 11179 (metadata registries). For more on ISO standards, see the reference: (International Organization for Standardization, 2026).
- *IETF*: Governs key web infrastructure such as URIs, MIME types, HTTP headers, and character encodings—all of which are essential technical metadata.
- *OASIS*: Maintains some metadata-related frameworks (often XML-based) that remain in use in enterprise environments.

In addition, when working with datatypes for data properties, follow the established XML Schema datatypes (e.g., `xsd:string, xsd:integer, xsd:dateTime`), which are defaults in tools like Protégé and cover the overwhelming majority of use cases. Unlike object properties, where domains often need specialized vocabularies, new datatypes are rarely needed. If you find yourself considering a new datatype, first ask whether it would be better represented as a class instead. For example, rather than inventing a custom `xsd:currency` datatype, model `Currency` as a class and relate it to values using existing numeric or string datatypes.

Rule of thumb: If Dublin Core or W3C has a standard for your technical problem, use it. If not, look to ISO or IETF before creating your own. For datatypes, reuse the standard XML Schema types and avoid inventing new ones unless absolutely unavoidable. In addition, you may find useful vocabularies in GitHub repositories or Linked Data sources. However, you should be far more cautious with such sources that haven't gone through the rigorous process that these standards groups require. To put it bluntly, there is a lot of junk out there, and starting with a vocabulary that is someone's MS thesis will often be more trouble than creating your own.

The Two Dublin Core Namespaces

For the vast majority of ontologies, it is a good idea to add basic metadata properties from Dublin Core. Dublin Core is probably the oldest technical vocabulary. It predates the Semantic Web, RDF, and OWL. It is one of those vocabularies that is essentially a de facto standard for metadata. However, when you look at the Dublin Core website, you'll find two different vocabularies. This can be confusing.

Because Dublin Core is so fundamental, I think it's worth explaining why there are two vocabularies and two namespaces and which you should use.

The Dublin Core Metadata Initiative (DCMI) originally released a small set of 15 metadata elements in the mid-1990s, using the namespace http://purl.org/dc/elements/1.1/.

This vocabulary is typically abbreviated simply as `dc:` and includes familiar properties such as `dc:title, dc:creator, and dc:date`. At the time, RDF and OWL did not yet exist, so these elements were defined informally without domain, range, or sub-property relationships. They were intended primarily for flat, document-oriented metadata used in web and document management systems. At the time, my team at Deloitte did a lot of work in content and document management, and we used Dublin Core for every project.

When RDF matured, DCMI introduced a richer namespace:

```
http://purl.org/dc/terms/
```

This newer vocabulary, known as DC Terms, redefined all the original elements in a semantically precise way and added many new ones (e.g., `dcterms:created`, `dcterms:issued`, `dcterms:modified`, `dcterms:license`). To maintain backward compatibility, every new `dcterms` property that corresponded to an older `dc` element was declared as a sub-property of that original dc: property. For example:

```
dcterms:creator rdfs:subPropertyOf dc:creator .
dcterms:relation rdfs:subPropertyOf dc:relation .
```

As a result, when you load the official DCMI ontology into a tool such as Protégé, the annotation property hierarchy appears baroque: `dc:relation` at the top, `dcterms:relation` as a sub-property, and additional specialized properties such as `dcterms:hasPart, dcterms:isPartOf`, and `dcterms:references` nested beneath that. This hierarchy was never intended for end-users; it was designed so that older metadata applications that only understood the legacy `dc` properties could still recognize data expressed with `dcterms`.

This structure causes confusion because for virtually all ontology work in OWL, the duplication between `dc` and `dcterms` adds no practical value. Recall that annotation properties are not reasoned over in the same way as object or data properties, so the sub-property hierarchy has no functional effect.

The `dc` vocabulary is effectively frozen; DCMI now recommends ***using only the*** *`dcterms`* ***namespace for all new work***. Maintaining both can lead to inconsistent data entry, where some users assert values on `dc:title` and others on `dcterms:title`. Unless you know of a specific backward compatibility issue you have where some old systems still need the older `dc` namespace, ignore `dc` completely and just use `dcterms`.

Where to Start

Standards documentation can be intimidating, and readers often encounter references to other specifications, profiles, or ISO standards with little explanation. You should not feel obligated to understand all of these layers upfront. In practice, most practitioners work with a small subset of vocabularies repeatedly and learn the rest as needed.

As a practical starting point, if you want to dig into some of these standards, focus first on Dublin Core. As described above, the Dublin Core Metadata Terms page is where you want to start (The Dublin Core Metadata Initiative (DCMI), 2024) rather than the legacy namespace provided for backward compatibility.

PROV-O (Prov W3C Working Group, 2013) provides a clear and widely adopted model for provenance: who created something, when, how, and from what. Provenance concepts recur throughout enterprise data architectures, data products, and governance, making PROV-O one of the most immediately useful metadata vocabularies.

Next, become familiar with SHACL (Knublauch & Kontokostas, 2017), not as a constraint language in the abstract, but as a practical tool for validating data and expressing design-time expectations. The SHACL Playground (Knublauch, 2026) is a good resource to experiment with defining SHACL models and using them to analyze RDF graphs.

For datasets and data catalogs, DCAT (Albertoni, et al., 2024) is the most important vocabulary to understand. It provides a standardized way to describe datasets, distributions, and access mechanisms and is increasingly used as the semantic backbone of data catalogs in enterprise environments. We'll go into a detailed Data Catalog ontology example that utilizes all of these standards in **Chapter 13**.

If you are interested in what I call semiformal models, then the Simple Knowledge Organization System (SKOS) is essential (W3C SKOS Working Group, 2005). Semiformal models are models such as taxonomies, lexical vocabularies, glossaries, and folksonomies. Many of these predate OWL, and creating OWL versions of these models utilizing SKOS can be an excellent way to add rigor to such models and to use those semiformal models as the first step toward ontologies. We'll discuss this in **Chapter 12**.

Other standards such as ISO metadata registries are best approached incrementally and only when a concrete use case demands them.

Domain Vocabularies

Unlike technical vocabularies, which are broadly applicable across domains, domain vocabularies are specific to an industry or scientific domain. As a result, there aren't a few organizations such as the W3C that you can rely on for domain vocabularies. The organizations and standards you should adopt depend directly on your domain of work.

Some of the most widely used domain vocabularies include

- *OBO Foundry Ontologies (OBO Technical WG, 2024)*: A large family of interoperable ontologies in the related domains of biology, healthcare, and life sciences (e.g., Gene Ontology, chemical entities of biological interest, oral health and disease). The OBO Foundry is one of the most successful examples of a large collection of interoperable ontologies that adhere to well-defined standards. It is a model for any organization developing their own collection of integrated ontologies.
- *SNOMED CT (SNOMED International, 2024)*: A comprehensive, standardized clinical healthcare ontology used worldwide.
- *FIBO (Financial Industry Business Ontology) (EDM Council, 2024)*: An Object Management Group (OMG) standard ontology for concepts in finance, banking, and business.
- *GoodRelations (München, 2025)*: A popular ontology for e-commerce and product information.
- *Schema.org (W3C Schema.org Community Group, 2026)*. A broadly used vocabulary for structured data on the web (e.g., products, events, organizations), supported by major search engines and Internet corporations.

For domain vocabularies, always look for vocabularies in your field that are backed by respected standards groups that include leading organizations in the domain or industry. These groups will require vocabularies to follow standards and pass quality metrics before being included. For example, FIBO is the global reference ontology for the financial industry, developed under the auspices of the Enterprise Data Management Council (EDM Association) and standardized with the Object Management Group (OMG). Its content has been reviewed and shaped by leading financial institutions, including Bloomberg, Citigroup, Goldman Sachs, Deutsche Bank, and Wells Fargo. Similarly, schema.org has the backing of Microsoft, Google, and other leaders in web hosting and applications.

There are a lot of low-quality domain-specific vocabularies in random websites and GitHub repositories. If a domain-specific vocabulary does not have the backing of a standards organization, review it carefully.

Upper Models

An upper model is different than a domain or technical vocabulary, although it often will have entities from both types of vocabularies. An upper model is a vocabulary meant to serve as a common foundation for all ontologies used in an organization or community. The idea is that by starting all ontologies with the same upper model, the ontologies will all share common best practices and definitions of common concepts used across the organization such as `Customer` or `Patient`. This is a very sound idea, and all organizations should standardize on an upper model. However, there can also be a cost to adopting upper models that are too broad: they may inhibit innovation and lock modelers into paradigms that are not appropriate for their requirements.

Some of the most widely used upper models in the academic world are based on philosophical ideas and concepts such as the Basic Formal Ontology (BFO) (Arp et al., 2015). Although these upper models are useful for academic communities, I encourage those in industry or who are building practical technology to avoid them. There are several issues with such philosophic upper models:

- *Making ontologies more difficult to understand*: Philosophic upper models use jargon that no engineer or business person will be familiar with such as "continuant" and "occurrent." This violates Gruber's clarity guideline (Gruber, 1993). Gruber's guidelines are relevant for all ontologies but even more so for upper models, which will be the starting point for ***every*** ontology in a community.
- *No proven benefit*: Philosophy and engineering have different goals and evaluation methods, and unlike good engineers, the philosophers who create upper models have never done any rigorous, double-blind, empirical evaluations that demonstrate that using their jargon makes ontologies easier to build, integrate, and reuse.
- *Imposition of unproven design constraints*: Philosophical upper models have constraints such as to not utilize multiple inheritance. This violates Gruber's rule of minimal commitment (Gruber, 1993). Again, while minimal commitment is a good design guideline for all ontologies, it is especially critical for an upper model that will establish constraints on ***all*** ontologies developed within a community.
- *Complex axioms with no value*: Philosophic upper models often have complex axioms based on the philosophical model behind the upper model. These axioms have no demonstrable value and immediately impose extra work for the reasoner on every ontology, before a single entity relevant to the business problem has even been defined. One of the reasons that practitioners often prefer Neo4j property graphs over Semantic Knowledge Graphs is performance. Adding complex axioms with no proven value is a practice that gives Semantic Knowledge Graphs a performance handicap from the very start.

These philosophical upper models aren't completely without value. Some of the best examples of curated high-quality vocabularies such as the Open Biological Ontology Foundry (Jackson et al., 2021) are built on such ontologies. In addition, there's much to be learned from many of their design guidelines for documentation and modeling. These frameworks are created by people who have spent decades thinking deeply about modeling problems, and some of their insights translate extremely well into practical OWL work.

For example, BFO has guidelines for natural language text that defines subclasses based on the restrictions they add to properties of superclasses. This was very useful when I needed to standardize documentation for a project. In another example, a project had to decide whether two "sub-part" properties in a domain ontology should share a common superproperty. BFO's treatment of parthood, and its clear separation of transitive from non-transitive sub-relations, helped us avoid arbitrary decisions and provided clear guidelines for making the appropriate decision.

These kinds of ideas—parthood, participation in events, identity, continuants vs. occurrents—can be genuinely helpful when you're facing a complex modeling decision, and resources and standards from OBO can be very useful. Some specific useful resources include OBO Principles for Good Ontologies (OBO Technical WG, 2024) and OBO Semantic Engineering Training (Overton et al., 2021).

It is an excellent practice for an organization to standardize on an upper model. However, that upper model should be lean and based on best practices. Most upper models focus on defining a common

semantics. However, metadata is as important to standardize. That is why I created the Basic Reusable Ontology (DeBellis, 2025) models by selecting properties and classes from some of the most important technical vocabularies: Dublin Core, SKOS, and PROV-O. In addition, a relatively new upper model that I find very useful is the Simplified Upper Level Ontology (SULO) (Michel Dumontier et al., 2025). Unlike many upper models, SULO is very lean and only provides common definitions for concepts that virtually all ontologies will require such as physical object. I've added this to the Basic Reusable Ontology models to provide a good basic foundation that is still very lean.

If a more expansive upper model is desired, the gist upper model (Blackwood, 2020) from Cambridge Semantics is a good starting point for many business domains. It has standard concepts such as Event, Person, and Place. Also, note that for many domains, the domain ontology itself serves as an upper model. For example, for ontologies that add semantics to websites, the schema.org vocabulary can provide an excellent upper model. For financial services, FIBO is both a domain model and an upper model.

One important principle to remember: reusing a vocabulary does not require you to reuse all of it. For example, when I use gist, I often remove axioms that I don't find useful. In the Basic Reusable Ontology models, I pruned out all the DC entities because as discussed earlier, they add complexity and are only required for rare cases of backward compatibility with pre-RDF systems.

We will next discuss details about how to import and manage vocabularies. There are two main ways to do this: using the catalog XML file and merging.

Importing Vocabularies

The starting point to reuse a vocabulary is to import it by its canonical IRI. For example, your ontology might declare

```
owl:imports  <http://www.w3.org/ns/prov#>
```

This indicates a formal dependency on PROV-O without copying its axioms into your file. In Protégé, you can do this in the **Active Ontology** tab. At the bottom of that tab, there are overlapping sub-tabs across the bottom of the tab: **Ontology imports**, **Ontology Prefixes**, and **General class axioms**. To import an ontology, click on the + next to **Direct imports** in the **Ontology imports** tab. This will launch a wizard that enables you to navigate your local file system or enter a URL for the ontology to import. After you do the import, the imported ontology will show up in that tab.

Figure 7-2 shows what the Ontology imports tab looks like after I imported the SKOS vocabulary into an existing ontology.

Once you have imported, you have two issues to deal with:

1. What happens if your Internet connection goes down or the site where you retrieved the vocabulary goes down?
2. Do you want to remain consistent with updated versions of the vocabulary?

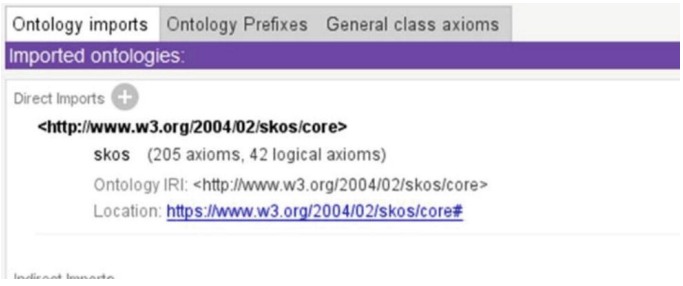


Figure 7-2 Ontology imports sub-tab in the Active Ontology tab

There are two ways to deal with these issues. Merging the vocabulary into your ontology is the easier way. However, using the ontology catalog is better if you want to be sure to get updates of the vocabulary, but it can be complex for new users. For details about how to use catalog files, see https://github.com/mdebellis/SemanticKG-Design/wiki/UsingCatalogFiles.

Merging

For small projects and quick prototypes, the level of effort required for using the catalog can be overkill. Also, when you reuse vocabularies such as Dublin Core, SKOS, and PROV-O, those vocabularies haven't changed in years and are unlikely to change in the future. For these types of projects, it makes sense to merge an external ontology directly into your own.

In Protégé, this can be done with the command **Refactor>Merge ontologies**. That command will launch a wizard that will ask you for the ontology to merge and the ontology to merge into. You want to merge reused vocabularies into your ontology. This creates a self-contained file with no imports. Merging brings the entities directly into your file so you don't need and don't want the import declaration, which will cause Protégé to reload the vocabulary. Therefore, you should delete the vocabulary from your imports after doing the merge (click on the **X** in the **Imported ontologies** tab).

Merging can be appropriate when

- You want a frozen snapshot for long-term archiving
- You are preparing simplified examples for teaching
- You prefer to just reuse an existing vocabulary and don't care about getting updates
- The vocabulary is stable and no longer updated
- You are working on a proof-of-concept prototype or other system where speed is essential and the code and environment are likely to go through major changes

For these types of projects, merging is simpler, faster, and gives you complete assurance that your ontology will have no issues from vocabularies not being available. Merging means that the reused vocabulary is just part of your ontology, just as entities you create yourself. Note: the merged entities *retain their external IRIs*, but they *never need to make any network connections* back to their original home.

Merging has trade-offs. Once merged, your ontology will no longer benefit from updates to the original. For most projects, modularity through catalog files (see the link above) is the better long-term strategy even though it takes a bit of effort the first time you do it.

Whenever you do a merge, make sure to add an `rdfs:comment` on the Active Ontology tab in Protégé that describes the specific vocabulary you are using, the IRI where you got it, the date you imported it, and the date you merged it. Do this ***before*** you do the merge, to make sure the changes are documented.

Ontology Patterns

As described above, OWL operates at the conceptual level of software engineering. Thus, analysis patterns from the object-oriented community are very relevant to the design of good ontologies. There are differences between OOP and OWL which we will discuss in more detail in **Chapter 9**. However, most of those differences are at the design level and hence aren't that significant for modeling OWL ontologies. The book *Analysis Patterns* by Martin Fowler has an incredible number of patterns that can be utilized with OWL. Of course, for the behavior part of those patterns, OWL doesn't provide a way to define behavior. However, the whole point of this book is to talk about ontologies that are part of a larger system, and such ontologies need to and do have ways to implement process as well as data as we'll discuss in **Chapter 9**.

The literature on ontology usually ignores the existing object-oriented patterns. The justification is that OOP and Description Logic are so different that object-oriented patterns aren't relevant to OWL. The

use of example patterns drawn from Fowler's book below demonstrates that this isn't true. Furthermore, any such argument is theoretically incorrect when applied to analysis patterns. The whole point of analysis patterns is that they are more abstract than, and not dependent on, any specific object-oriented programming language. In fact, as I will demonstrate below, if anything, OOP analysis patterns are in some ways easier to model in OWL. Logical constraints and definitions can be formally modeled and reasoned about in OWL but can only be described in text or graphics in pattern languages such as the one Fowler uses.

Another common approach in the ontology literature is to define new languages for ontology patterns. This is somewhat analogous to the discussion Fowler has in a more recent book on *enterprise patterns* regarding the distinction between Java Objects and Enterprise Java Beans (Fowler, 2002). Fowler points out that many of the services people think require EJB components can be done just as well (in fact, more simply and with less overhead) with simple Java objects. The same is true regarding ontology patterns. OWL is a very high-level language. Logic and set theory are as abstract as you can get. Patterns can be defined as simple OWL entities. There is no need for new pattern languages.

Agent (Party)

The key idea behind this pattern is that there are many types of relations and data that can apply to a person, an organization, or an AI agent. They can be assigned tasks, they can be held accountable for events, they have information that describes their physical and virtual address, etc.

This is the first pattern in Fowler's Analysis Patterns book (Fowler, 1997), although he calls it a Party (as in "party to a contract"). It is used in Friend of a Friend (FOAF), PROV-O, and almost every OWL ontology I've ever built. Figure 7-3 shows the graph for the pattern as it is used in the People Ontology and visualized in Gruff. The one difference between this pattern and the standard one used by Fowler, FOAF, and PROV-O is that I've added `AI_Agent` as an additional subclass of `Agent`.

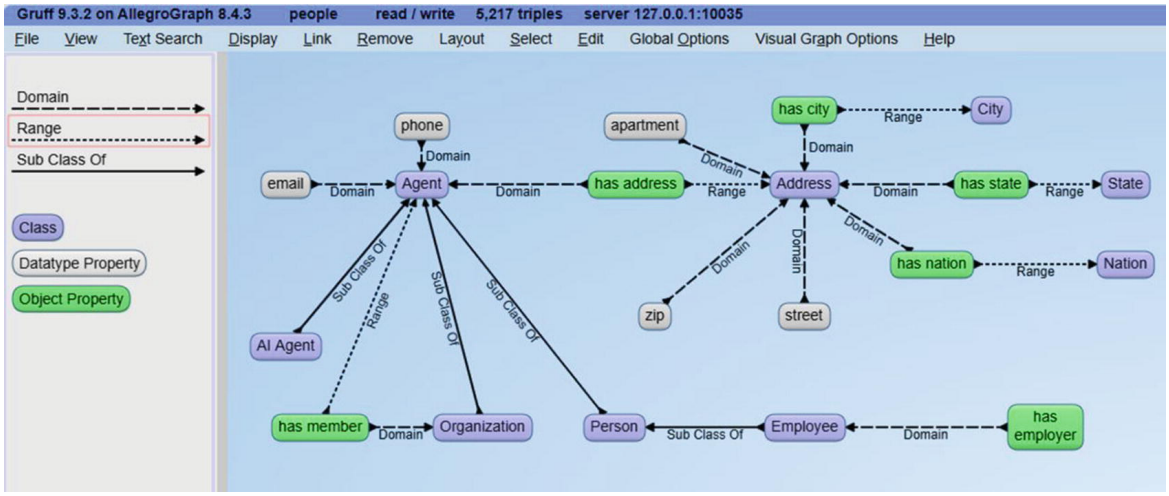


Figure 7-3 Agent pattern in Gruff

The Agent pattern is also a good example of a heuristic for cleaner modeling regarding classes and properties. Whenever you see two or more classes used repeatedly across several domain or range definitions, it's cleaner to explicitly define a class that those classes are a subclass of. In ontologies which have failed to do this, it is common to see domain and range definitions that amount to specifications of a set, e.g., `'Person or Organization'`. In general, if you start to see definitions like, say, `'Class_Foo or Class_Bar'` recurring in the domain or range of your properties, that is often an indicator that you should define a common superclass of whatever concepts `Class_Foo` and `Class_Bar` happen to be. That superclass can then be referenced throughout. This makes your ontology

easier to understand, because the overarching concept has been made explicit. It also makes your ontology easier to maintain. Updates to the model need only be made in one place rather than many, and the reasoner will take care of propagating implications of the change.

Also, recall the concept of an anonymous class. When you use expressions like `'Person or Organization'`, you are creating an anonymous class that is the superclass of `Person` and `Organization`. Since you are creating an anonymous class anyway, it can often be better to model such a class (e.g., `Agent`) explicitly rather than use the anonymous class created by the reasoner.

N-ary Relations (Property Graphs)

A major, valuable feature of property graphs, which OWL lacks, is the ability to allow properties to themselves have data of their own. Property graphs do this by representing the property as an object, rather than implicitly as a binary relation as in RDF and OWL. The n-ary relation pattern is the mechanism for achieving the equivalent effect in OWL.[1] Remember that relation is just another name for a property in OWL. An n-ary relation is capable of storing data on the actual relation, rather than simply having the relation map from one object to another. A classic example is modeling employment. The simplest way to model employment is illustrated in the People Ontology: a property called `has_employer` with domain `Person` and range `Agent` with an inverse property `is_employer_of`.

However, this simple model soon breaks down if we want to model that a `Person` can have more than one employer at a time and that almost everyone will have more than one `Employer` over the course of their life. In these situations, if we want to keep track of information such as the `start_date`, we can't do it. With a property graph, we can record information such as the start date and title directly on the link. OWL is constrained to binary relations to make reasoning more efficient. However, there is an easy design pattern that allows us to model the same concept in OWL. What we do is reify the relation. The word "reify" is formed from res (Latin: "thing, object") + the suffix -fy (from Latin: "to make"). So, reify literally means to "thingify" or "make into a thing," and that is exactly what we do in OWL. A binary relation in OWL isn't explicitly represented. It is implicit in the value of the property on an instance that points to another instance. To reify the relation, we make a new class that explicitly models the relation. In this case, `Employment`. Then, we change the domain and range accordingly. Properties such as `start_date` are now asserted on the `Employment` object rather than on the `Employee`. The `has_employer` property has domain `Employment` now rather than `Person`, and we create a new property `has_employment` to model the relation between a `Person` and an `Employer`.

Figure 7-4 shows the data for von Neumann in the People Ontology. We know he worked for IBM and the Institute for Advanced Study, and we have two start dates, but there is no way with this simple model to associate the appropriate start date with the appropriate employer.

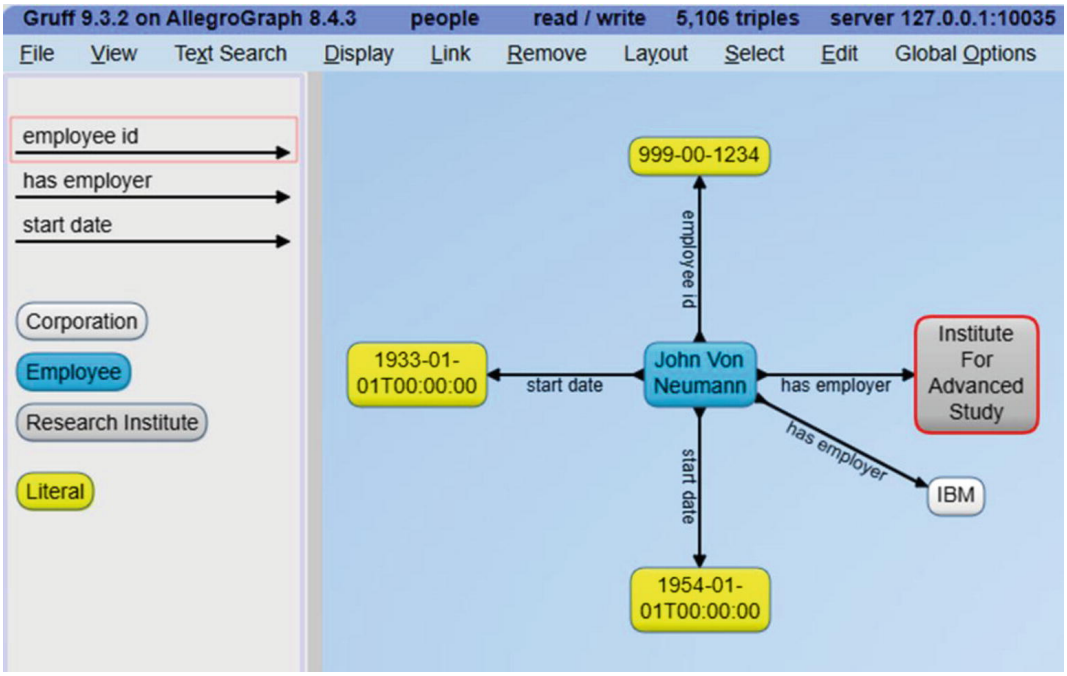


Figure 7-4 Example instance data illustrating need for n-ary relation pattern

Figure 7-5 shows the revised model after we applied the n-ary relation pattern. Now we have a class called `Employment` that can model the specific dates when the employee started work as well as their other info such as their employee ID.

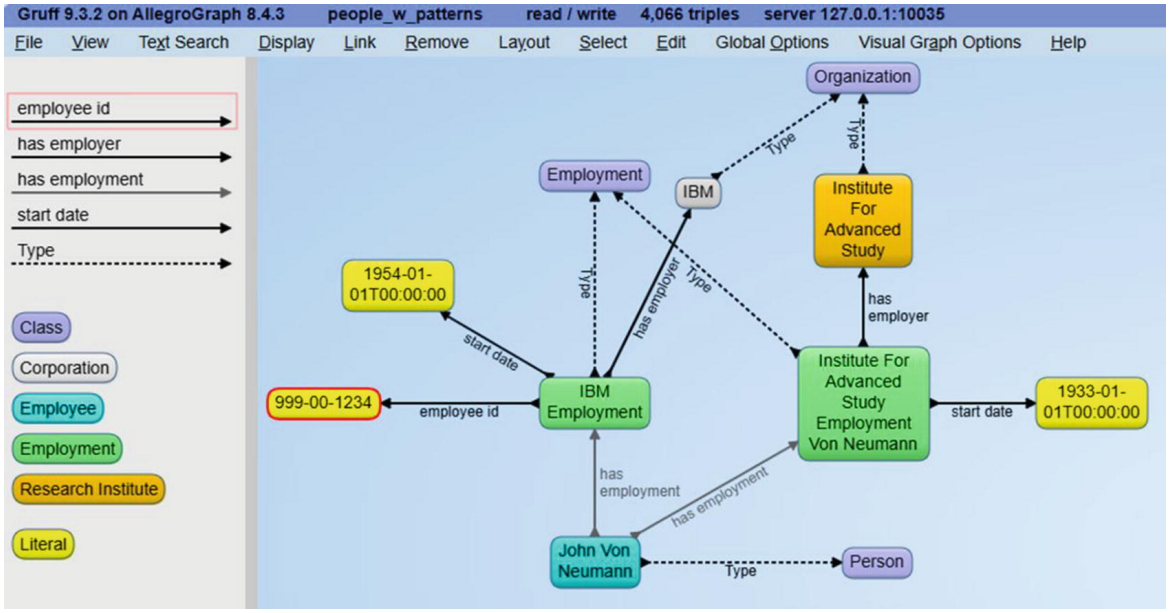


Figure 7-5 Example instance data after n-ary relation pattern

The Collection Pattern: When to Reuse vs. Differentiate Properties

Earlier in this chapter, we described three different types of buckets, one of which was the collection bucket. Collections are everywhere in modeling: cities in states, employees in organizations, parts in machines, voters in political parties. They all share a family resemblance, but not all collections are the same.

Broadly, we can distinguish several kinds of "collection buckets":

- *Containment*: Physical or spatial inclusion. Example: Vancouver is contained in British Columbia.
- *Component of*: Structural parts of a whole. Example: A CPU is a component of a computer.
- *Membership*: Individuals belonging to a group. Example: An employee is a member of an organization.

The challenge for ontology engineers is deciding when to reuse a single property across different collections and when to define distinct properties.

If two collections share the same logical semantics, they can reasonably use the same property. For example, employees in a corporation and members of a political party can both be modeled with a general property such as `has_member`. The semantics are consistent: both capture membership in a group.

If two collections have different ontological semantics, then separate properties should be defined.

Containment and components are not the same as membership. California does not "have members"; it has cities and counties that are contained within it. A car does not "have members"; it has components that make up its structure. Using the same property would blur important distinctions.

The Basic Formal Ontology (BFO) (Arp, et al., 2015) offers a useful principle here:

- Parthood (e.g., part of) has sub-relations that distinguish structural components (engine/car) from spatial containment (Vancouver/British Columbia).
- Membership is modeled separately, since it relates an individual to a group rather than a part to a whole.

Guideline: Use the same property if the relationship is a variation on the same semantics of membership or grouping. Define different properties if the relationships reflect fundamentally different kinds of collection: containment, component, or membership.

In practice, it is tempting to overgeneralize and use one property such as `part_of` everywhere. Doing so sacrifices clarity and can mislead downstream reasoning. By recognizing the different kinds of collection "buckets," you can choose properties that preserve the semantics of each relationship while still following a consistent pattern.

Another useful aspect of the collection bucket is to define a transitive super property for collections that have nested structure such as containment or components. In the People Ontology, we have `has_component` and `has_component_transitive`. The first property gives us the direct relation where one has another object as a direct component. For example, `Motherboard has_component CPU`. The transitive super property will use the reasoner to compute all the components for an object. In the simple computer example used in the People Ontology, `has_component_transitive` gives all the components and the components of those components that define a simple model of a computer.

Ordered Value Partition

This pattern is used when you have instances that model some quality that, while it can't be expressed as a number, still has a rank ordering. This is defined as an enumerated class and with properties where the reasoner does most of the ordering for you. This is clearly easier than having to explicitly model the n*(n-1) rank relationships between every element in the class.

The `Role` class in the People Ontology is an example of an ordered value partition. `Technical_Role` is a standard way to model the type of work an `Employee` does on a team. In the People Ontology, we defined `Technical_Role` as an enumerated class that is `{Director, Manager, Scientist, 'Senior Staff', Staff}`. We have two properties that model the concept of seniority: `has_higher_seniority_than` and its inverse `has_lower_seniority_than`.

We made these properties transitive with respect to each other. This significantly reduces the effort required to define the ordering. We only need to define the order of each instance going in one direction.

The reasoner will infer the rest for us. For example, in Figure 7-6, we can see all the assertions (highlighted in yellow) made by the reasoner on Manager, derived from these four assertions:

```
Director has_higher_seniority_than Scientist
Scientist has_higher_seniority_than Manager
Manager has_higher_seniority_than Senior_Staff
Senior_Staff has_higher_seniority_than Staff
```

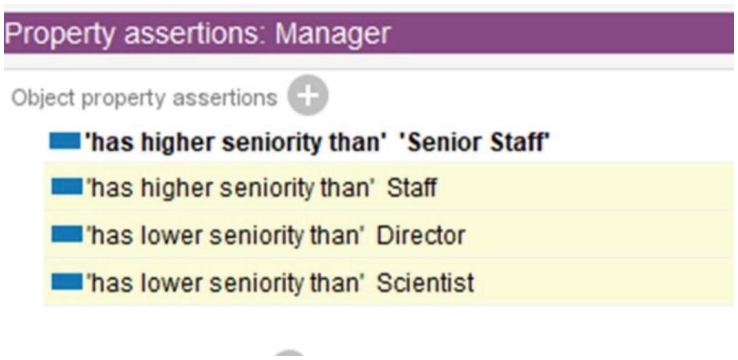


Figure 7-6 Example of ordered value partition pattern

Those are all the assertions we need. The reasoner will infer the rest. In some cases (e.g., ordering processes in a queue), we may want to also track the immediate predecessor and successor task. In that case, we would add a transitive super-property to each property that defines the order and remove the transitive characteristic from the sub-properties.

The Measurement Pattern

This is another pattern that goes back to OOP. This pattern is used when you want to model measurements with a unit of measure (such as meters for distance, minutes for time, grams for weight). This seems mundane, which perhaps is why this is one of the most common patterns that people often neglect. Rather than explicitly model units, it's very tempting to simply decide upon (but not explicitly model) conventions that are supported by software ("We'll just remember that age is modeled in years, distance in meters, etc."). This makes things easy in the short term, but risks confusion later when—not if, when—someone invariably enters data while assuming different units.

A well-known example from space exploration illustrates the danger of leaving units implicit. In 1999, NASA's Mars Climate Orbiter was lost because one software component produced thrust data in pound-force seconds (lbf·s), while another component interpreted those same values as newton-seconds (N·s). The numerical values were syntactically valid and passed through the system without error. However, because the units were not modeled explicitly and validated automatically, the spacecraft entered the Martian atmosphere at the wrong trajectory and was destroyed.

The lesson is not about metric vs. imperial systems; it is about semantics. When measurement units are not explicitly represented in a model, software systems can operate correctly according to their own assumptions and still fail catastrophically. Explicit modeling of measurements is a safeguard against errors that simple data typing cannot detect.

There are existing unit vocabularies that can be utilized for units: QUDT (Ray, 2025) or OM for physical units; UCUM codes are excellent as identifiers; ISO 4217 for currencies. However, the QUDT ontology has a lot of baggage; it covers much more than units, and for many applications. I've found it easier to implement the pattern as demonstrated in the People Ontology.

The pattern relates two classes: subclasses of `Measurement` (e.g., `Distance_Measurement`, `Time_Measurement`) and subclasses of `Unit` (e.g., `Distance_Unit`, `Time_Unit`).`Measurement` has an object property called `has_unit` which must be an instance of the appropriate `Unit` class and a

data property called `value` that is an `xsd:decimal`. Any subclass of Measurement that is created inherits (and presumably further specifies) this object property.

Figure 7-7 illustrates how a distance measure using meters as the default unit is defined.

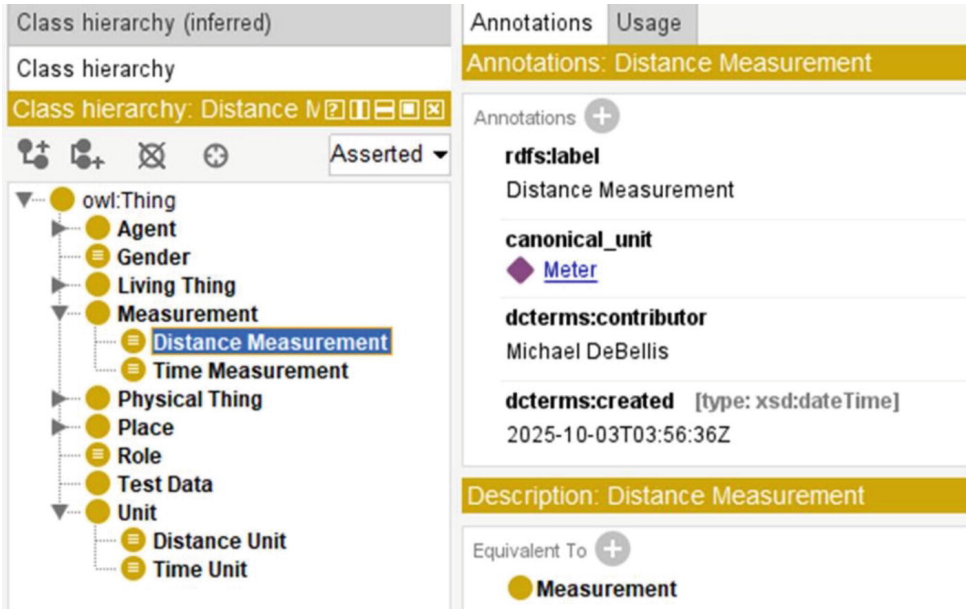


Figure 7-7 Classes utilizing the measurement pattern

In addition, there are object properties: `greater_than`, `less_than`, and `equal_to` that have domain and range of `Measurement`. The properties `greater_than` and `less_than` are inverses and transitive. There needs to be some code or rule that performs conversion and ordering. For example, that can compute that 200 centimeters is `less_than` 5 meters.

In the People Ontology used in this book, I've added some simple SWRL rules to put distance measurements in the appropriate order and have created some example distance measurements. This is just a small example for instructional purposes. The approach used in this example establishes a canonical value for each type of measurement. For example, for distance, the canonical unit is meters. This is stored as an annotation property on each class, as seen in Figure 7-7. First, SWRL converts each measurement to its canonical value. Then, a single SWRL rule can determine for all distance measurements if one is greater than the other. Once this is determined, the reasoner can compute the `less_than` inverses. It is tempting to add an axiom that `equal_to` is reflexive. Any measurement is equal to itself. However, as stated above, this will result in the domain and range of `equal_to` being defined as `owl:Thing` and an assertion for each object that it is equal to itself. This adds a lot of clutter even to a small knowledge graph such as the People Ontology. Instead, we simply add another rule that if two measurements measure the same thing (the class of their unit is the same) and they have the same `canonical_value`, then they are `equal_to` each other.

EXERCISE 7-2: ADD RULE AND DATA FOR MEASUREMENT PATTERN

The Measurement pattern is a good example to illustrate how to do math in SWRL. There is a library of predefined operators in SWRL for math, dates, strings, etc., that can be found at (Horrocks et al., 2004). To see how these work, open the People Ontology in Protégé and select: **Window➤Tabs➤SWRL Tab**. That will take you to the GUI to create and edit SWRL rules. You will see that there are several rules such as Canonical_Distance_Centimeter_Conversion.

These rules convert distance measurement values to the canonical unit which is meters. Click on the centimeter conversion rule and select **Edit** from the buttons that show up in the lower right corner

beneath where the rules are displayed (slightly to the right of the tab's middle). You aren't going to edit this rule, just view it. That will take you to the following rule for converting centimeter measurements to the canonical meter value. SWRL sometimes changes the order of the left side of a rule. It has no effect on the execution so the order of the predicates may look a bit different, but what matters is that all of them should be there:

```
Distance_Measurement(?m1) ^ has_unit(?m1, Centimeter) ^ value(?m1, ?v1)
^ swrlb:divide(?cv, ?v1, 100) -> canonical_value(?m1, ?cv)
```

Since the canonical unit is meters, we need to divide the value for centimeters by 100 (100 centimeters = 1 meter) and store that new value as the canonical value. The `swrlb:divide` operator is one of the predefined SWRL operators for math. As with all SWRL operators that create new values, the way it works is that it performs the operation on the second and third values (in this example, it divides the `?v1` value by 100) and binds that new value to the first variable which must be a new unbound variable. In this example, it divides the value by 100, binds that value to the new unbound variable `?cv`, and places the resulting number as the `canonical_value` of the distance measurement. Note that the implementation of this would require many rules for distance as well as the other types of units so as with all of the People Ontology there are just a few example rules and values so you can see how the conversion and ordering work.

Select **Cancel** to close the window viewing that rule without changing anything. Then, look for the `Distance_Ordering_Rule` and select it and choose **Edit** (although you are also going to just view it). That rule (again order of left-hand predicates may differ) is

```
Distance_Measurement(?m1) ^ Distance_Measurement(?m2) ^
canonical_value(?m1, ?v1) ^ canonical_value(?m2, ?v2) ^
swrlb:greaterThan(?v1, ?v2) -> greater_than(?m1, ?m2)
```

This compares the `canonical_value` of every distinct pair of instances of the `Distance_Measurement` class and for each case asserts that the one with the `larger canonical_value` is represented as `greater_than` the other measurement. Select **Cancel** to close the window viewing that rule without changing anything.

If you review all the rules, and the various distance units, you'll see that we have the unit value `Mile` but no rule to convert it to the canonical value in meters. Let's fix that by creating a new rule. We could click the **New** button and write the whole rule; however, there is an easier way to do it (and this is something that comes up often when writing SWRL rules). What we want is an existing rule that is similar so we can copy that rule and just make a few changes.

Since a mile is greater than a meter, we are going to multiply the value to get the canonical value. Let's start with a rule that already does that but for a different distance unit. Kilometers are also greater than meters, so select the `Canonical_Distance_Kilometer_Conversion` rule. However, don't edit it, but instead select the **Clone** button. That will create a copy of that rule with the default name `S1`. Find, select, and edit the `S1` rule. The rule should look similar to this when you start:

```
Distance_Measurement(?m1) ^ has_unit(?m1, Kilometer) ^ value(?m1, ?v1) ^
swrlb:multiply(?cv, ?v1, 1000) -> canonical_value(?m1, ?cv)
```

We want to change the `Kilometer` unit to `Mile` and the 1000 in the multiply statement to 1609.34 which is the number of meters in a mile. Make those changes, rename the rule from `S1` to `Canonical_Distance_Mile_Conversion`, and **Save** the new rule.

We aren't done yet. There are no current distance measurements with units in miles. Let's create one. Go to the **Individuals by class** tab. If that tab isn't one of your current tabs, use

Window➤Tabs➤Individuals by class to create one. There should be a **Class hierarchy** view in the upper left side of that tab. Expand the Measurement class and select the Distance_Measurement class. You should see in the view below the class view there are several distance measurements that already exist. If you select each one, you should see on the **Property assertions** view in the lower right corner that each one has a canonical value and they are all in the appropriate order except for the instance labelled `1 Mile` that only has a value but no unit. Select that instance. In the lower right corner, there should be a view labelled **Property Assertions: 1 Mile**. Right now, it should only have one property assertion that its `value` data property is 1.0. Select the + next to **Object property assertions**. That provides you with an editor for the label of the object property and the value. In the first (leftmost) field type: 'has uni then hit <control><space>.

That should auto-complete to 'has unit'. Note you need to type in that much because there is also an object property called 'has uncle', although if you just typed 'has u and did <control><space>, you would just get a menu with the two properties that match and you could select the unit property that way.

In the second field, for the value, type Mil and then <control><space>, and it should auto-complete to `Mile`. Note: in the Protégé UI when doing auto-complete on entities, if the entity has a label with a space in it, you need to put the label name starting with a single quote. If the label doesn't have a space (like `Mile`), you can just start typing and hit auto-complete without the quotes around it.

In the very lowest right-hand corner of Protégé (this is independent of what tab you are in), there should be a message that says: **Reasoner state out of sync with active ontology**. Do **Reasoner➤Synchronize reasoner**. Be sure you are using the Pellet reasoner because it supports SWRL rules the best. Also, there is a little check box next to that message labelled **Show inferences**. It should be checked. There are seldom times you wouldn't want it checked, but sometimes it is possible to accidentally uncheck it. If you aren't seeing inferences produced by the reasoner at any point, that is always the first thing to check. The 1 Mile instance should now be in the correct order with all the other distance measurements and should look like Figure 7-8. Note that all the greater than and less than and equal to values have been filled in automatically by other rules. That is the beauty of a good pattern; once you set it up, it is easy to extend, and the reasoner does most of the work.

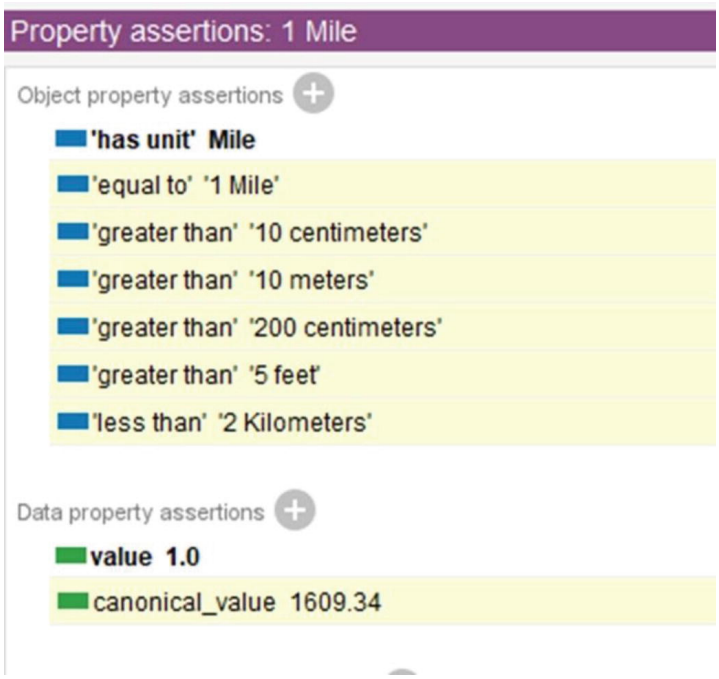


Figure 7-8 Example instance using the measurement pattern

Using Sample Instances in Conceptual Modeling

Most approaches to conceptual modeling emphasize defining classes and properties first, and only later introducing individuals. In practice, however, creating sample instances early in the process offers significant advantages, even at the purely conceptual stage. I recommend that, at a minimum, you create an instance that will test the various options for each axiom. For example, if you have an axiom that an `Adult` is a `Person` whose `has_age` is greater than or equal to 18, at a minimum, create three test instances: a `Person` who `has_age` > 18, a `Person` who `has_age` = 18, and a `Person` who `has_age` < 18.

Early Error Detection with the Reasoner

Reasoners often reveal modeling mistakes only when there are individuals to test against. For example, disjoint class violations or overly restrictive cardinalities may remain invisible until specific cases are introduced. One of the basic concepts of software engineering is that the sooner an error is found, the cheaper it is to fix. Adding sample data ensures many types of errors that the reasoner can detect are caught as early in the life cycle as possible.

Bottom-up Design

Instances are not just about testing; they stimulate thinking. Creating a concrete example often sparks realization of missing properties, new class distinctions, or alternative patterns of representation that a purely abstract TBox model might miss.

Modeling real examples highlights gaps. Trying to represent that von Neumann was employed by Princeton and IBM quickly exposes that the ontology can't support multiple employments. These insights often emerge only when working bottom-up from individuals.

Even in conceptual modeling, sample instances are required. They anchor the design in real-world use cases, provide immediate feedback from the reasoner on problems that would otherwise be overlooked, and facilitate both top-down and bottom-up refinement.

WebProtégé

Modeling is a team sport, which makes WebProtégé an excellent supplement to Protégé (Tudorache, Tania, et al., 2020). Unlike the desktop version, WebProtégé allows concurrent design and includes additional tools such as threaded discussions that facilitate collaborative design.

Figure 7-9 illustrates an example of the People Ontology with a comment and reply I created to simulate collaboration between a domain expert and an ontology designer. In the first comment, I played the role of a domain expert pointing out that the instance `John_Von_Neumann` illustrates a flaw in the current design. He was employed by the Institute for Advanced Study and IBM. However, there is no way given the modeling of employment in the People Ontology to model which properties such as `start_date` and `employee_id` were associated with his employment at IBM vs. Princeton. In the reply, I played the role of an ontologist, suggesting that the proper solution to this problem is to utilize the n-ary relation pattern and create a class called `Employment` that explicitly models the information for different employers.

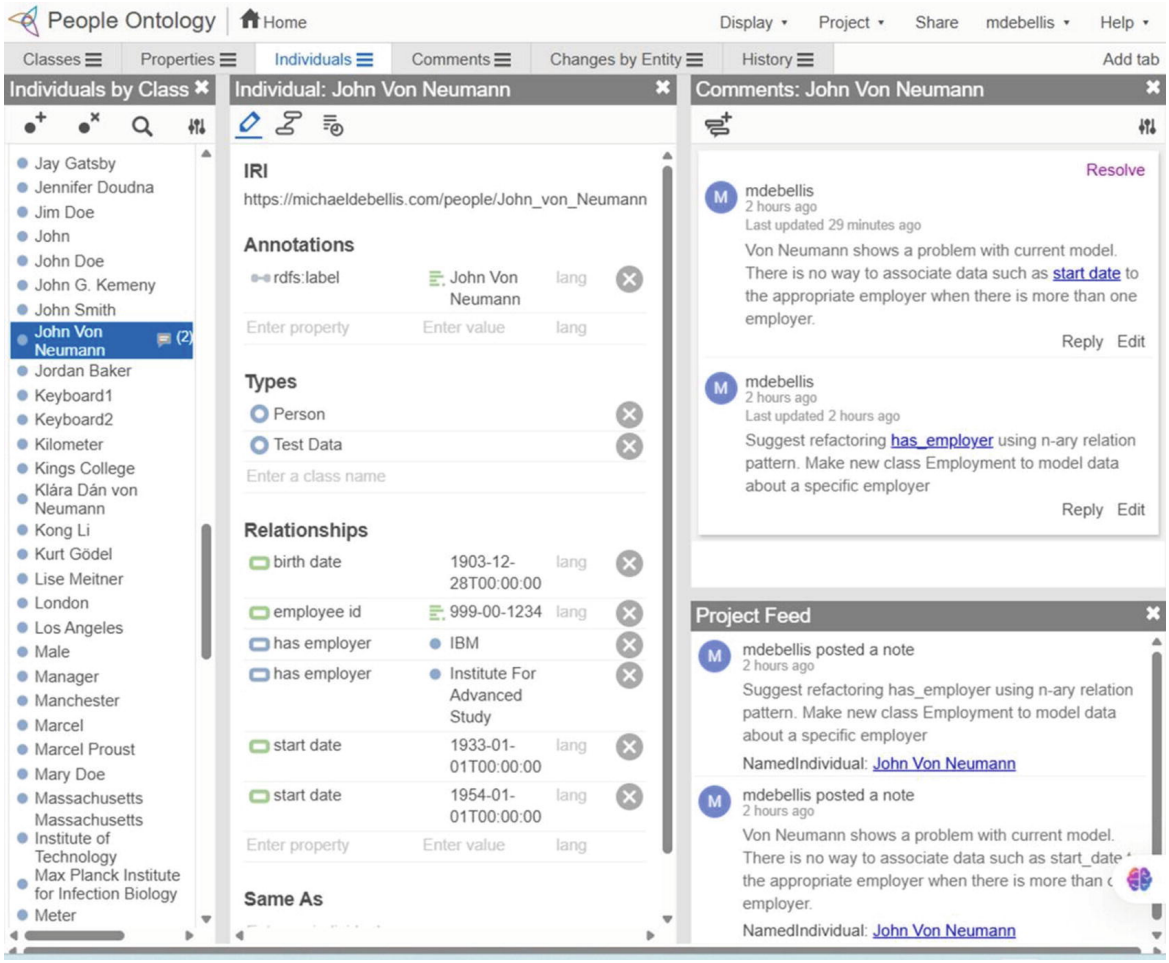


Figure 7-9 WebProtégé provides collaborative editing tools such as threaded discussions

To see the example in Figure 7-9 in WebProtégé, go to the URL https://tinyurl.com/PeopleJVonNeumann. Note: you will have to create an account in WebProtégé first. You can do that here: https://webprotege.stanford.edu/. Once you have an account, the TinyURL above should take you to the People Ontology opened on Von Neumann. You can also browse the ontology in general and see the differences between Protégé and WebProtégé.

Figure 7-9 shows WebProtégé after the People Ontology has been refactored to utilize the n-ary pattern.

Note that while WebProtégé is an excellent tool for group design, it should never be used on its own because it does not currently support a reasoner. Using the reasoner from the very start of a project is essential to get the full power of OWL. One of the most common problems I see with ontologies designed by new users is that they wait until they have defined a significant part of the ontology before running the reasoner. What inevitably happens is that there will be several logical errors in the ontology. When this happens, the reasoner typically can't offer any useful guidance as to what the problems are. That is why the reasoner needs to be run early and often. When there is only one logical error, the reasoner can point you right to it and the problem will be obvious and easy to fix.

Luckily, WebProtégé facilitates "round-trip engineering." Users can export the ontology from WebProtégé, import it into Protégé or a graph database, use reasoners and other tools, make changes to the ontology in those tools, and then reload the new ontology back into WebProtégé. WebProtégé not only keeps all the additional data such as threaded discussions but in its history view, it shows the

changes made in other tools as if they had been made in WebProtégé. For more guidance on how to best utilize WebProtégé, see **Chapter 12** WebProtégé in DeBellis (2021).

WebProtégé is best utilized in the early stages of conceptual modeling, when the team is working out basic issues such as the main classes and properties. In these activities, WebProtégé can function as the primary copy of the ontology and one of the technical team members can make sure to correctly define the IRIs and regularly take the ontology from WebProtégé, load it into the desktop version, run the reasoner, and fix any errors detected by the reasoner. Once the initial foundation of classes and properties is defined, the team can then load the ontology into GitHub and use it to control versioning, branching, etc.

Using LLMs to Define Conceptual Models

LLMs are useful for all aspects of model development, data design, testing, and implementation. There are essentially three major ways I've used LLMs (mainly ChatGPT) to help develop conceptual models:

1. Use the LLM as a subject matter expert.
2. Use the LLM to add TBox definitions based on websites and other authoritative documentation.
3. Use the LLM to add ABox definitions based on the ontology itself as well as external sources.

However, LLMs have to be used with care. Two things in particular to keep in mind when using an LLM are that their answers are very dependent on the details of their prompts and that LLMs are typically designed more to engage you than challenge you. Just today, I was having a dialog with ChatGPT about adding AI Agent as an additional subclass (besides Person and Organization) to Agent. I accidentally typed "`AI_Agent` as a subclass of Person". Rather than saying, "That is a bad model!" it went along with what I said and gave me a model based on that (obviously terrible) design constraint. This is what leads some people into delusions that, with the help of an LLM, they've solved the problem of dark matter or created a new physics that reconciles relativity and quantum theory. Always remember to ask for criticisms rather than validation. Ask it to find potential problems. That helps get the most out of an LLM and avoids having it reinforce and extend something in your prompt that may be problematic.

Using an LLM As a Subject Matter Expert

One of the most common issues when developing an ontology is what's called the Knowledge Acquisition Bottleneck. This is the inherent tension that makes it difficult to get the time of the people who have the best understanding of the domain. Although this has its own name for rule-based and ontology systems, in my experience, this problem applies to any kind of system. The people who understand the business the best are too busy running the business to spare time to share their expertise.

This is one of the reasons it can be difficult for teams to embrace Agile. Agile requires changing the culture and taking the valuable time of scarce resources as full-time team members. That isn't an easy cultural change, and every time I've helped clients try Agile, there is always great skepticism from some that it is worth taking these people away from their day-to-day duties. However, in my experience, while I've often seen clients struggle with this change, I've never seen a client go back once they try Agile.

Agile succeeds where other methods fail because the initial requirements are seldom correct without full-time participation of the business experts. Agile also succeeds because people hate change; it's just a fact of human nature. Thus, a big problem in software engineering is training the users on the new software. With Agile, the users are part of the team and are pulling the new software rather than pushing against it.

An LLM is not a substitute for a human expert. However, when a human is unavailable, an LLM can often provide a developer with some circumscribed value. It can give answers to questions about the

domain that still have to be validated by an expert, but this may be quicker for the expert than starting from zero. The same is true for guidance as to how to model the domain. In both cases, the developer can make progress and what time the subject matter expert has can be used more efficiently.

Using an LLM to Define Conceptual Entities

For adding data to a knowledge graph with an LLM, there are two ways one can do it:

1. Take as input the existing knowledge graph (e.g., as a Turtle file) and then generate a new Turtle file with the additional data.
2. Generate a SPARQL query which simply adds the new data to the graph using an INSERT DATA update or use options such as selecting a class and using Protégé commands such as **Tools➤Create class hierarchy.**

I've found that the second approach works far better than the first. The reason is that in the first approach the LLM will have to keep both the complete existing ontology as well as the changes in its memory. For any decent-sized ontology, this will be too much data for the LLM to keep in its memory. The result is the LLM will leave out some of the data. With the second approach, we ***focus only on the new data*** which dramatically reduces the memory requirements. In addition, we can focus even more by asking for specific kinds of data and creating prompts for each type of data that will yield several INSERT DATA updates for each type of data.

Even when generating `INSERT DATA` updates, the amount of data can overwhelm the LLM. For example, I used an LLM to generate first Spanish and then English labels for an ontology that had no labels (DeBellis, 2024a). For translating the Spanish labels, I had to divide the work into smaller chunks of `INSERT DATA` updates (e.g., define English labels for classes, then for object properties, etc.).

One way this can be useful is adding TBox axioms (classes, properties, and axioms that define them) by using sources from the Internet and documentation. In some recent work I did, I wanted to model all the various industries. The focus of the ontology was Europe, so I had a dialog with ChatGPT about modeling industries in Europe. It pointed me to the Industry Classification Benchmark (ICB) from FTSE Russell (LSEG).

In this example, rather than using `INSERT DATA`, I asked ChatGPT to generate an indented list of classes and subclasses that corresponded to the ICB model. Then, I selected the `Industry` class and used the **Tools➤Create class hierarchy** option and pasted the indented list from ChatGPT. The ICB model has approximately 170 industries with an average of four levels of nesting from industry sectors to subsectors. In a few seconds, I was able to add the entire model to my ontology and without worrying about the inevitable errors that would have occurred if I had added the classes manually.

Using an LLM to Define Test Data Instances

When designing test data for an ontology, it is important not only to create examples that demonstrate the intended inferences but also to exercise edge cases that might reveal mistakes in the axioms. A useful analogy comes from software engineering, where Boundary Value Analysis (BVA) is a well-established testing technique. In BVA, the input space is divided into valid and invalid equivalence classes, groups of values that the program should treat in the same way, and test cases are chosen that fall on and around the boundaries of these partitions. An equivalence class, in this sense, is simply a way of partitioning data according to shared conditions. This is similar to defined classes in OWL: a defined class has necessary and sufficient conditions that *partition* individuals into the class when those conditions are met. That is why, in Protégé, such conditions are entered under the **Equivalent To** field in the **Class Description** view. For example, if an ontology defines `Adult` as a `Person` who `has_age` ≥ `18`, then the equivalence classes are three sets of `Person` with boundary values at 17, 18, and 19. By generating test instances at these boundaries, along with cases such as a person with no age value, to test an edge condition, we can

verify that the reasoner correctly classifies individuals into the intended defined classes, while also revealing potential unintended inferences. In this way, reasoning over OWL axioms becomes directly comparable to executing test cases against the specification of a software system.

Another practice I recommend is to make all the test data an instance of a class called `Test_Data`. This makes it trivial to remove the test data when you start ingesting real data and makes it clear to anyone reviewing the ontology when data is for testing and does not come from actual systems.

For example, to create test data for the People Ontology, I uploaded the file to ChatGPT and then prompted it by saying:

> *Generate test data instances for this ontology as an INSERT DATA SPARQL update. Focus only on test data for the Person class and its subclasses based on axioms related to :has_age to keep the size of the test data reasonable. Make each instance an instance of the* `Test_Data` *class as well as any domain classes. Follow our standard naming conventions for IRIs. Generate test data that tests the various boundary conditions for each axiom. For example, if an axiom defines an Adult as a Person whose age is >= 18, generate instances of Person that test each possible edge condition: age > 18, age < 18, and age = 18.*

Since I often use ChatGPT to generate Semantic Knowledge Graphs, it already knows that "our standard naming conventions" mean that classes and instances should start with capital letters, that underscores are used for blank spaces, and that properties are all lowercase. This prompt generated the SPARQL shown in Listing 7-1.

```
PREFIX :      <https://michaeldebellis.com/people/>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:   <http://www.w3.org/2002/07/owl#>
INSERT DATA {
############################################################
# Adult ≡ Person ∧ has_age ≥ 18
# Child ≡ Person ∧ has_age < 18
# Testing strategy: Equivalence Partitioning + Boundary Value # Analysis
############################################################
# EP invalid partition (age < 18) + BVA edge just below
:Alice_Person_17 a :Person, :Test_Data ; :has_age "17"^^xsd:integer .
  # EP valid partition (age ≥ 18) + BVA exact boundary
:Bob_Person_18   a :Person, :Test_Data ; :has_age "18"^^xsd:integer .
# EP valid partition (age ≥ 18) + BVA just above boundary
  :Carol_Person_19 a :Person, :Test_Data ; :has_age "19"^^xsd:integer .
# EP edge case: no value at all (Open World Assumption test)
  :Dana_No_Age     a :Person, :Test_Data .
############################################################
# Boy ≡ Child ⊓ has_gender Male
# Girl ≡ Child ⊓ has_gender Female
# --- EP partitions by gender; BVA age < 18
############################################################
# Valid partition: Child + Male → should infer Boy
  :Eddie_Age12_Male    a :Person, :Test_Data ; :has_age "12"^^xsd:integer ;
:has_gender :Male .
  # Valid partition: Child + Female → should infer Girl
```

```
  :Fiona_Age12_Female  a :Person, :Test_Data ; :has_age "12"^^xsd:integer ;
:has_gender :Female .
  # Partition outside definition: Child + Non_Binary → should infer Child,
but not Boy/Girl
  :Noah_Age12_NonBinary a :Person, :Test_Data ; :has_age "12"^^xsd:integer
; :has_gender :Non_Binary .
#################################################################
# Man ≡ Adult ⊓ has_gender Male
# Woman ≡ Adult ⊓ has_gender Female
#################################################################
  :George_Age30_Male   a :Person, :Test_Data ; :has_age "30"^^xsd:integer ;
:has_gender :Male .
  :Hannah_Age30_Female a :Person, :Test_Data ; :has_age "30"^^xsd:integer ;
:has_gender :Female .
  :Indy_Age30_NonBinary a :Person, :Test_Data ; :has_age "30"^^xsd:integer
; :has_gender :Non_Binary .
#################################################################
# Functional property: has_gender (ObjectProperty)
# --- EP: single gender (valid) vs multiple genders (invalid)
#################################################################
:Alex_DualGender a :Person, :Test_Data ; :has_age "25"^^xsd:integer ;
                   :has_gender :Male, :Female .
:Male owl:differentFrom :Female .   # force inconsistency
#################################################################
# Functional property: has_age (DatatypeProperty)
#################################################################
:PolyAgePerson a :Person, :Test_Data ;
                 :has_age "21"^^xsd:integer ,"22"^^xsd:integer.
#################################################################
# Employee ≡ Person ∧ has_employer some Agent
# Self_Employee ⊑ Employee
#################################################################
:IBM_Corp a :Corporation, :Organization, :Test_Data .
# Valid partition: employer is Organization (Agent)
:Shannon_Employee a :Person, :Test_Data ; :has_employer :IBM_Corp .
# Edge case: employer is a Person (still an Agent, but less typical)
:Einstein_Person a :Person, :Test_Data .
:Eve_EmployeeToPerson a :Person, :Test_Data ; :has_employer
:Einstein_Person .
# Self_Employee case
:Sam_SelfEmployed a :Self_Employee, :Person, :Test_Data ;
                    :has_employer :Sam_SelfEmployed .
#################################################################
# Spouse: Symmetric +
Irreflexive  #################################################################
# One-sided assertion → should infer symmetry
:Grace_Hopper   a :Person, :Test_Data ; :has_spouse :Vincent_Foster.
:Vincent_Foster a :Person, :Test_Data.
# Optional violation (BVA for irreflexivity)
:Grace_Hopper :has_spouse :Grace_Hopper .
```

```
############################################################
# Parents: has_father (range Man), has_mother (range Woman)
############################################################
:Child_X  a :Person, :Test_Data ; :has_father :Father_X ; :has_mother
:Mother_X.
  :Father_X a :Man, :Test_Data.
  :Mother_X a :Woman, :Test_Data.}
```

Listing 7-1 SPARQL Test Data Generated by ChatGPT

Using Generated Test Data

The best way to test this is to

1. Load the People Ontology into AllegroGraph
2. Use Gruff to run the `INSERT DATA`
3. Export the ontology and call it something like `people_ontology_test_data.ttl`
4. Load that test data ontology into Protégé and run the reasoner

For more on loading and saving knowledge graphs with AllegroGraph, see https://github.com/mdebellis/SemanticKG-Design/wiki/Exercises.

If you don't want to use AllegroGraph, I also have included a version of the People Ontology in the GitHub repository that already has the test data loaded: https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/people_ontology_test_data.ttl.

When you load the ontology with errors into Protégé, you will see three errors. The latest version of Protégé handles multiple errors better than in the past. Each time the reasoner should give you information to find each error so you can correct it, run the reasoner again, then find and fix the next error. For more details, see https://github.com/mdebellis/SemanticKG-Design/blob/main/ontologies/readme.md

---

Differences in Coping with OWL Reasoner Errors

You may be wondering: why not run the reasoner in AllegroGraph? The reason that won't show the errors is that the AllegroGraph reasoner is meant for much larger graphs. Hence, it is forgiving when you have a graph with logical errors. It doesn't make the entire model inconsistent; it just asserts the improper value. This may seem inconsistent, but it is actually appropriate.

Protégé is a modeling tool. It is not meant to handle the huge amount of real-world data that AllegroGraph and other graph databases work with. You want Protégé to show you logical errors in your model that test data will expose. However, when you get to the level of needing a graph database, you should be using SHACL (which as we will show in the next chapter can be generated from OWL axioms) to validate your data, not the reasoner.

Summary

- *Bucket is a general term for different ways to model categories of related entities*. There are several ways to model buckets: instances of a class, collections, and tags.

- *Multiple inheritance is a powerful modeling tool* where a class can have more than one superclass, and an instance can be an instance of more than one class.
- *An important modeling heuristic is to remember that by definition a model is an abstraction of reality.* When we model, we ignore some aspects of reality in order to focus on others.
- *Ontology reuse is an excellent practice.* It can save time, provide teams with the knowledge of experts captured as ontologies, and make Semantic Knowledge Graphs easier to integrate with other systems and knowledge graphs.
- *Ontology patterns are common designs that can be used to solve many different kinds of problems.*
- *Instances must be included in an ontology from the very beginning*. Instances can help the reasoner discover errors, can serve as test data, and can improve the modeling process.
- *Large Language Models (LLMs) can be very useful when modeling*. Some of the things they can do include serving as a subject matter expert, helping create initial conceptual models, and generating test data instances with property values.
- *Further Reading*. The revised Pizza Ontology tutorial provides in-depth hands-on examples for utilizing Protégé (DeBellis, 2021). *Demystifying OWL for the Enterprise* by Michael Uschold is an excellent book for more information on conceptual modeling with OWL (Uschold, 2018).

Footnotes

1 To be precise, n-ary relations provide additional features that property graphs don't. Property graphs can only add attributes (a.k.a. data properties) to a property, but they can only model one relation (object property). The n-ary relation pattern can model both additional relations and attributes (additional object and data properties).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_8

8. Design Modeling: From Meaning to Robustness

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

In **Chapter 7**, we focused on defining the conceptual domain model using OWL classes, relationships, individuals, and logical constraints. In this chapter, we move from that conceptual foundation to a design model that can be implemented, validated, and integrated into real-world data pipelines. By the end of this chapter, you will understand how to move from a conceptual ontology to a design model that is grounded in real-world constraints, supports validation and automation, and integrates effectively with modern data architectures.

The chapter begins by defining a design model and how it differs from a conceptual model. Then, I'll describe some of the design concepts that are most relevant for Semantic Knowledge Graphs, such as the CAP theorem and vertical vs. horizontal scaling. Both RDF graphs and other types of conceptual graphs are crucial for understanding how to create design models for Semantic Knowledge Graphs. Thus, just as **Chapter 3** had a minimal primer on set theory and logic, I'll provide a simple overview of key concepts from the mathematical discipline of graph theory.

With that background, I'll then discuss named graphs in RDF and how they can be utilized to define encapsulation and security boundaries. Next, I'll discuss data ingestion and show an example of ingesting new data using the Python Prefect tool. This is followed by a discussion of SHACL including how to generate SHACL from OWL and why OWL is primarily for conceptual modeling and SHACL for design modeling.

This will be followed by a discussion of the most common types of transformations used with Semantic Knowledge Graphs and how the ROBOT tool from the OBO Foundry can automate design activities such as transformations and testing.

The last two topics are internationalization and OWL profiles. Internationalization is more important than ever, as support for multiple languages is becoming the norm in many systems. It is considered a design activity because it doesn't change the conceptual domain model but rather how that model is exposed to users. Finally, I will conclude with a discussion of OWL profiles. There are several official, and even more unofficial, vendor subsets of the OWL language that trade off expressive power for efficiency. Selecting the appropriate profile is an essential design task.

What Is a Design Model?

The design model marks the transition from conceptual domain modeling to concrete decisions about how a Semantic Knowledge Graph will be built, validated, and operated. In the data lifecycle introduced in **Chapter 4**, this is the stage where conceptual models encounter real data, real systems, and real constraints.

Where the conceptual model focuses on meaning and correctness in the abstract, the design model focuses on *robustness*: how the system behaves in the presence of incomplete, inconsistent, or simply wrong data. At this stage, the central question is no longer "What *should* be true?" but rather "*What will inevitably go wrong, and how do we prevent those failures from disabling the system?*"

OWL, SWRL, Protégé, and WebProtégé are essential tools for conceptual modeling. At the design level, however, SHACL, SPARQL, the capabilities of the underlying graph database and external tools for data ingestion are important. These technologies allow us to express data integrity constraints, validation rules,

and transformation logic in ways that acknowledge the messiness of real-world data. This shift in emphasis from *idealized correctness* to *controlled imperfection* is the defining characteristic of design modeling.

Before examining the mechanics of design modeling in detail, it is important to understand the broader architectural context in which Semantic Knowledge Graphs operate. In practice, design decisions are rarely made in isolation. They are shaped by the surrounding data ecosystem, including relational databases, streaming systems, and analytic platforms.

The transition from conceptual modeling to design modeling represents a shift from describing a domain to describing the real-world constraints that the model must adhere to. While the conceptual model defines meaning, the design model defines how that meaning is realized in practice. The key shift from conceptual to design modeling is that these real-world constraints are rigorously defined and automatically enforced through validation, processing, and system behavior.

Another way to understand the shift from conceptual to design modeling is through what software engineers have traditionally called nonfunctional requirements or, more informally, the "ilities." These include properties such as reliability, performance, latency, fault tolerance, scalability, and security.

In the conceptual model, our goal is to model the domain and its solutions as abstractly and accurately as possible, independent of technical constraints. The emphasis is on meaning: what concepts exist, how they relate, and what must logically hold.

In the design model, those concerns remain important, but they are no longer sufficient. We must now account for how the system behaves under operational conditions: how it responds to invalid data, how it scales as data volumes grow, how quickly it can answer queries, and how resilient it is in the face of failure.

Whereas logic and set theory were the most important formalisms behind conceptual modeling, the most important formalism for design is graph theory. Hence, we next provide a very brief overview of the different types of graphs defined in mathematical graph theory. Understanding these basic concepts will provide a foundation for the rest of the chapter.

A Minimal Graph-Theory Primer for Semantic Knowledge Graphs

Graph theory is a topic in mathematics. Its terminology provides a useful foundation for understanding Semantic Knowledge Graphs. This perspective is especially important because the Semantic Web is, by design, graph-native: RDF, SHACL, and even more abstract languages like OWL ultimately reduce to graph structures. In addition, workflows such as the data ingestion example below are typically described as a type of graph. As a result, it is useful to understand some basic concepts from mathematical graph theory.

In **Chapter 3**, we introduced the minimal logical background needed for OWL-based conceptual models. Here, we take the same approach for graphs: we introduce only the graph-theoretic ideas required to understand design-level decisions. The same caveats apply here as to our discussion of logic and set theory: the math-phobic have nothing to fear. There won't be any proofs or anything that requires deep mathematical knowledge or reasoning. We'll just become familiar with some common kinds of graphs so that in the rest of the book and in your work, when you see terms like tree or directed acyclic graph (DAG), you know what those terms mean.

A *graph* is defined as a set of nodes and edges that connect the nodes. A *directed graph* is a graph where a direction (left to right, up to down) has significance in the meaning of the graph such as the passage of time or going from general to specific concepts. A labelled graph is a graph where the edges or nodes (or both) have labels.

In a directed graph, a node that is downstream of another node is a child of that node and the upstream node is the parent of the child node. For example, if a graph shows the passage of time going from left to right, all nodes that are directly connected to a node X and are to the right of that node are the *children* of node X and X is the *parent* of those child nodes. An *orphan* is a node that has no children or parents.

The *root* nodes in a directed graph are nodes that

1. Are at the "beginning" of the graph. That is, if the directed graph shows a class hierarchy with subclasses below their superclasses, the class or classes at the top with no superclasses are the root(s).

2. Have child nodes but no parents.

A *tree* graph is a special kind of graph that satisfies three conditions:

1. It is a directed graph.
2. It has a single root.
3. Each node in the graph (except the root) has exactly one parent.

An example of a tree graph is a subclass hierarchy that starts at owl:Thing and doesn't use multiple inheritance.

A *network* graph is the least constrained type of graph. In a network graph, pretty much anything goes. There are semantics to the connections between nodes, but there is no semantic meaning in the direction of the graph.

The last basic concept to understand is a cycle. A cycle is a path such as A → B → C → A. Acyclic means that there is no path that starts and ends at the same node. In other words, you cannot have a sequence like A → B → C → A in an acyclic graph.

Now that you know the basics of graph theory, let's drill down to the implementation level. How does all this apply to graphs in the TBox? We'll start with those graphs because they are more constrained than the ABox, and thus, a simpler entry point to this topic.

Graph-Theoretic Implications for TBox Modeling

OWL class and property hierarchies are directed graphs. The direction reflects the relationship between a subclass and its superclass, or between a sub-property and its super-property. For example, the tuples that define `has_husband` form a subset of those for `has_spouse`, so the edge `has_husband` → `has_spouse` is directed from a more specific property to a more general one.

OWL class and property hierarchies each have a single root. For classes, `owl:Thing;` for object properties, `owl:topObjectProperty`; and for data properties, `owl:topDataProperty`. Because of this, class and property hierarchies that use single inheritance form trees. Property hierarchies seldom use multiple inheritance, but it does happen, especially for object properties. As discussed above, multiple inheritance is a powerful modeling tool and is often used for class hierarchies.

When class or object hierarchies use multiple inheritance, they are *directed acyclic graphs (DAGs)* rather than trees.

It is possible to define cycles in OWL class and property models. For example, you could say that Supervisor is a subclass of Manager, and Manager is a subclass of Employee, and then, perhaps as the result of a subtle modeling mistake, assert that Employee is a subclass of Supervisor. Written out:

```
Supervisor ⊑ Manager
Manager ⊑ Employee
Employee ⊑ Supervisor
```

These axioms form a cycle: each class is (directly or indirectly) a subclass of the next. Because the cycle leads back to the starting point, this hierarchy is a cycle.

OWL 2 DL reasoners infer that all the classes participating in a cycle must be equivalent, since each one is both a subclass and a superclass of the others. In this example, the reasoner would conclude

```
Supervisor ≡ Manager ≡ Employee
```

This is usually a sign that something is wrong with the model. Cycles are allowed syntactically in OWL class hierarchies, but they cause the reasoner to collapse the classes involved into a single equivalence class,

which is usually not what the modeler intended. If you see that the reasoner has turned classes that you consider different such as `Supervisor` and `Employee` into the same class (this is shown in the Protégé UI), that is often a sign that you have a cycle. Usually, you can detect the cycle by just inspecting the classes that the reasoner has made equivalent. However, the tools that find anti-patterns (ROBOT and OOPS!) in ontologies discussed in **Chapter 5** will also find cycles.

The same is true for cycles in property hierarchies although they are much less common because property hierarchies are almost always trees.

To sum up the relevant graph theory for the TBox:

- Property hierarchies are virtually always trees.
- Class hierarchies with only single inheritance are trees.
- Class and (rare) property hierarchies with multiple inheritance are DAGs.
- Cycles are allowed in class and property hierarchies but are virtually always unintended and evidence of a modeling error.

Now we move on to graphs that can be generated by connecting instances via property values. These graphs are much less restrictive.

Graph-Theoretic Implications for ABox Modeling

The most basic RDF triple is a directed graph. It is directed because each RDF triple (subject, predicate, object) is a directed edge from subject → object, where the predicate acts as the edge from one node to another.

The TBox (the class and property hierarchies) has a very strict graph-theoretic structure. They are always a tree or a directed acyclic graph (DAG). The ABox is very different. The ABox contains the individuals in the ontology and the object property assertions that connect them. When we visualize these instance-level connections, we are no longer dealing with trees or DAGs. Instead, ABox models often form the most general kind of graph, the network graph.

A network graph is simply a graph with no restrictions on its structure. Nodes may have any number of incoming or outgoing edges, relationships may form arbitrary patterns, and cycles are not only allowed but extremely common. For example, we assert in the People Ontology that

```
:Elsa_Einstein:has_spouse :Albert_Einstein
```

And since `:has_spouse` is symmetric, the reasoner infers that

```
:Albert_Einstein :has_spouse :Elsa_Einstein
```

This instance graph includes a two-node cycle. This is perfectly fine for instance-level data. The reasoner won't infer that Elsa and Albert Einstein are the same person.

Because most object properties describe natural relationships (e.g., married to, lives in, employed by, collaborates with), the instance data often forms richly interconnected networks rather than a clean branching structure. Two scientists may work at the same institution, live in the same city, appear in each other's publications, and influence each other's work; each of those connections contributes another edge to the network. For this reason, instance-level semantic graphs rarely are trees and often are networks.

The Relationship of Graph-Theoretic Issues to Tool Functionality

Part of why the above dive into graph theory was so necessary is that it's essential to understanding the services—and limitations—afforded by key tools. This section briefly reviews those considerations.

SPARQL queries extract subgraphs of this larger instance-level network. For example, a query asking for all scientists and their spouses returns a small portion of the overall graph, but it still obeys the same "anything goes" network structure. SPARQL does not impose hierarchical constraints; it simply returns all matching triples, which may form trees, DAGs, cycles, clusters, or densely interconnected networks.

Visualization tools such as Gruff, Stardog Explorer, and GraphDB's Visual Graph display these SPARQL result graphs in various layouts. When these tools visualize the TBox, they often show a tidy tree or DAG because the underlying structure is hierarchical. But when they visualize ABox data, the resulting graph is typically a full network. Typically, these graphs have cycles, cross-connections, and clusters reflecting the real-world complexity of the domain.

This contrast between the TBox and the ABox is fundamental:

- TBox graphs are highly constrained (trees or DAGs; cycles usually are an error).
- ABox graphs are unconstrained (general networks; cycles allowed; arbitrary structure).

Understanding this difference helps ontology engineers interpret SPARQL visualizations correctly and avoid expecting instance-level graphs to exhibit the clean hierarchical structure found in class and property hierarchies. These basic graph structures show up repeatedly in Semantic Knowledge Graph practice.

DAGs also appear in another important part of Semantic Knowledge Graph practice: data pipelines. Many workflow systems such as Apache Airflow represent the flow of data processing steps as a DAG of activities, where each activity depends on the outputs of its predecessors.

One final point from graph theory is worth mentioning: the more structure a graph has, the easier it is to display it in a readable way. This is another reason why class and property hierarchies (trees and DAGs) are straightforward to visualize, while instance-level networks are not. General networks often have many cross-connections, cycles, and local clusters, and arranging such graphs so that they are legible is a genuinely hard computational problem.

Modern tools such as Gruff, Stardog Explorer, and GraphDB's Visual Graph hide this complexity. They use sophisticated layout algorithms that position nodes and edges in ways that minimize views where many edges intersect or where labels from one edge may obscure those of another. These algorithms are solving a difficult problem, and the fact that these tools work as well as they do on large instance graphs is something we now take for granted.

Many visualization tools also let you choose alternative layouts, such as tree layouts, radial layouts, or force-directed layouts. Even when the underlying instance graph is not a tree in the strict graph-theoretic sense, a tree-like layout can be helpful when exploring data from a particular starting point or perspective. The key idea is that instance-level graphs are flexible, unconstrained networks, and different layouts can highlight different aspects of the data. We will see this in later chapters. For example, in **Chapter 12**, we will build a small ontology that can be visualized as a tree even though it covers several different kinds of properties.

This brief graph theory overview provides a shared vocabulary that will help make the rest of this chapter comprehensible. We now turn from graph structures in the abstract to the role Semantic Knowledge Graphs play in real architectures. These include integration with relational databases, trade-offs in distributed system architectures, data ingestion, and using subgraphs to define design boundaries.

The Role of Semantic Knowledge Graphs in Real Architectures

Semantic Knowledge Graphs must coexist with relational databases, columnar analytic stores, streaming platforms, and domain-specific operational systems.

This is true across a wide range of use cases. A Semantic Knowledge Graph may be used to catalog and describe existing data assets, to define conceptual and technical models for future systems, to show the relations (synonyms, similar to, broader, narrower) between lexical vocabularies,[1] or to support applications that leverage LLMs such as retrieval-augmented generation. In all of these cases, however, the underlying data is usually sourced from systems built around SQL models.

Despite periodic predictions to the contrary, SQL-based systems are not disappearing. Modern analytic platforms such as Google BigQuery, Snowflake, and Redshift, as well as streaming technologies such as Kafka, rely on the relational model to store, process, and analyze data. Designing a Semantic Knowledge Graph without acknowledging this reality leads to models that may be elegant in theory but of limited practical value in practice.

From a design perspective, the question is therefore not whether Semantic Knowledge Graphs should replace relational systems, but how they should complement them. Relational and columnar databases excel at large-scale aggregation, numerical analysis, and high-throughput querying. Semantic Knowledge Graphs excel at representing meaning, active metadata (Zaidi & Thoo, 2019), integrating heterogeneous data sources, managing schema evolution, and supporting reasoning across domains.

Design modeling is where these roles are made explicit. Decisions about what data is ingested into the knowledge graph, what remains in external systems, and how the two are linked, through identifiers, mappings, and metadata, have significant implications for performance, scalability, and maintainability. These are called nonfunctional requirements because they do not define what the system must do, but rather how well the system must perform: how fast it must respond, how much data it must be able to handle, and what rules for security it must adhere to.

For this reason, many successful Semantic Knowledge Graph deployments (e.g., Data Catalogs) treat the knowledge graph not as a replacement for existing analytic infrastructure, but as a semantic layer that sits alongside it: organizing, contextualizing, and connecting data that continues to be stored and processed in other systems such as relational databases.

These architectural realities introduce constraints that directly influence design choices. Among the most important of these is the CAP theorem, which highlights the trade-offs inherent in distributed systems.

Design Constraints from Distributed Systems (CAP and Scaling)

System design is all about making the right trade-offs.
—Alex Xu (Xu, 2025)

We begin by describing the most fundamental concepts for efficient design of distributed data:

1. The CAP theorem for distributed systems
2. Vertical vs. horizontal scaling
3. Using sharding to improve performance

The CAP Theorem

The CAP theorem (Gilbert & Lynch, 2002) is an example of how architecture design is driven by trade-offs. The CAP theorem states that a distributed system cannot simultaneously guarantee *consistency*, *availability*, and *partition tolerance*. A network partition error occurs when components of a distributed system temporarily cannot communicate with each other. When a database is distributed across multiple machines, network partitions are unavoidable. This requires designers to prioritize consistency vs. availability when planning for such failures.

If the system prioritizes availability, each partition continues operating independently, but without a complete view of the system's current state. Consider a system where data is distributed across multiple nodes in different locations. If the system is designed to prioritize availability, it must return a response to every request, even during a network partition failure. In this case, a query may return data from a local node that has not yet received the most recent updates, resulting in stale or inconsistent results. For example, a dashboard showing real-time metrics may display slightly outdated values because the system cannot wait to synchronize across all partitions. If the system instead prioritizes consistency, it must ensure that all nodes agree on the most recent state before responding. During a network partition, this may require delaying or rejecting requests until the system can confirm that the data is up to date. For example, a financial transaction system may refuse to process a request if it cannot guarantee that account balances reflect the latest updates across all nodes.

In practice, this means that systems handling analytic or observational data often favor availability, while systems responsible for critical updates or transactions tend to favor consistency.

This is not merely an operational concern. The choice directly affects what kinds of answers a system can safely return. A system that favors availability may return incomplete or temporarily inconsistent data. A system that favors consistency may reject requests until a coherent global state can be restored.

From a design perspective, CAP is best understood as a constraint that must be acknowledged explicitly. Different design models assume different positions within this trade-off, whether consciously or not.

Vertical and Horizontal Scaling

Scaling strategies determine what to do when CAP trade-offs become unavoidable. Vertical scaling (scaling up) increases the capacity of a single machine by adding memory, CPU, or storage. Because the system remains logically centralized, consistency is straightforward to maintain. Many Semantic Knowledge Graph systems favor vertical scaling for as long as possible, as it preserves global reasoning and semantics.

Horizontal scaling (scaling out) distributes data and computation across multiple machines. While this enables systems to grow beyond the limits of a single node, it introduces network communication as a fundamental dependency. *Once a system scales horizontally, CAP trade-offs drive design.*

Sharding and Graph Databases

Sharding is a specific technique to implement partitioning. It is the most common mechanism used to enable horizontal scaling. The term *partition* refers to the general concept of splitting data into pieces. Sharding partitions data across machines according to some key or strategy that attempts to group *data that is likely to be used together* in the *same partition*. For example, one might group all the data about a customer into the same shard. While sharding works well for many tabular and analytic workloads, it introduces unique challenges for graph-structured data.

In relational databases, sharding typically partitions rows based on a key, and most queries can be answered within a single shard or through well-understood join strategies. In contrast, graph databases encode meaning primarily through relationships and those relationships often span large portions of the graph. As a result, sharding a graph risks fragmenting its topology: traversals may cross shard boundaries, partial failures may yield structurally incomplete results, and global reasoning or inference may require additional time. For Semantic Knowledge Graphs, where correctness depends on a coherent global model, these effects are more severe than simple performance degradation.

Vendors address this challenge in different ways. There are three strategies that databases typically take regarding sharding:

1. *Hidden Sharding*: This is the simplest strategy. Sharding is done automatically and is not exposed to or controlled by the user. In spite of its simplicity, this approach can often be very efficient. For example, tools such as MongoDB and Google's BigTable both use this strategy and can scale with little degradation in performance.
2. *Strong Consistency*: This strategy emphasizes strong transactional consistency across clusters, constraining distribution across shards in order to preserve query correctness. Systems that prioritize strong consistency are more commonly used in operational contexts, such as financial or transactional systems, where correctness is critical. In contrast, analytic systems are often more tolerant of temporary inconsistencies and may prioritize availability and performance.
3. *Federated Sharding*: This strategy distinguishes between physical sharding for scalability and logical federation for semantics, allowing architects to distribute storage while preserving explicit semantic boundaries. AllegroGraph is an example of the latter approach: it supports sharded distributed repositories for scaling while also providing federation as a first-class mechanism for composing multiple repositories under a unified query interface (Franz, Inc., 2025). This strategy gives the user the most control but is also the most complex.

Design Implications for Semantic Knowledge Graphs

These considerations lead to an important conclusion: design strategies for knowledge graphs are often significantly different than for relational or columnar databases. Graph databases that prioritize semantic correctness and reasoning typically choose consistency over availability, while graph systems designed for massive scale relax consistency to remain available under partition.

Ultimately, design choices are driven by the specific requirements for each unique system. However, there are some general heuristics that describe the different design strategies appropriate for different databases. These are summarized in Table 8-1.

Table 8-1 Scaling and CAP Implications on Relational, Column, and Graph Databases

Relational databases were designed for strong consistency on a single node and scale vertically very well; while they can be sharded, doing so complicates joins and transactional guarantees, making horizontal scale-out difficult although achievable with care. Columnar databases are a recent innovation to efficiently store analytic data. Example columnar databases are Google BigQuery, Snowflake, and Amazon Redshift. Columnar databases organize data by columns rather than rows, making them especially well-suited for large analytic queries that scan selected fields across many records. Columnar databases are used when organizations need to analyze very large datasets, up to petabytes, without scanning every field of every row.

Graph databases face an even harder problem: because meaning is encoded primarily in relationships, sharding risks fragmenting graph topology, making traversal, inference, and global consistency semantics-sensitive and costly to preserve. In contrast, columnar databases are highly optimized for aggregation queries over large datasets and perform best with denormalized or star-schema designs. While they do support joins, complex join-heavy workloads that require low-latency, transactional access are generally less efficient than in row-oriented or graph-based systems.

Subgraphs as Design Boundaries

Another important issue in design modeling is encapsulation. From a design perspective, subgraphs are a primary mechanism for encapsulation in a Semantic Knowledge Graph. Just as modules or packages provide boundaries in software, subgraphs provide boundaries inside a graph database. They are central to scalable design.

As Semantic Knowledge Graphs grow, we quickly reach the point where we can no longer think of an individual knowledge graph as a single amorphous mass of triples. Different teams own different parts of the data. Some triples represent the ontology model (TBox). Instance (ABox) data includes domain data, provenance, triples created by the OWL reasoner, and other metadata. We need a way to partition this space without losing the benefits of a unified graph.

RDF provides this capability through subgraphs, usually implemented as named graphs. Intuitively, a named graph is just a set of triples grouped under an IRI. The collection of all these graphs together is called an RDF Dataset. SPARQL operates over such datasets and gives us a way to talk about "which graph" we are querying or updating.

In RDF 1.1, a dataset consists of a default graph together with zero or more named graphs. Each named graph is a set of RDF triples. Graph membership applies to *triples*, *not to RDF resources*. As a result, an RDF node (a class, property, or individual identified by an IRI) may appear in triples across multiple named graphs, while each individual triple belongs to exactly one graph. This distinction allows different triples about the same resource to be distributed across graphs for different purposes, such as separating core domain models from governance metadata, provenance information, security annotations, or inferred knowledge. Named graphs therefore support modularity and separation of concerns without fragmenting the underlying conceptual model.

What the Standards Specify (and What They Don't)

The RDF 1.1 and SPARQL 1.1 specifications (the current ones as I write this) define the concept of an RDF Dataset as

- A default graph (unnamed)
- Zero or more named graphs, each identified by an IRI

A SPARQL query lets us refer to these graphs using the `GRAPH` keyword. The SPARQL operations `CREATE GRAPH`, `DROP GRAPH`, `CLEAR GRAPH`, `MOVE`, `ADD`, and `COPY` allow us to do all the standard CRUD (create, read, update, delete) operations on graphs.

However, the standards do not currently prescribe

- How graphs are stored internally
- Whether graphs are created implicitly when you first insert into them
- How empty graphs are handled
- How graph-level security or access control works
- What administrative APIs are exposed beyond SPARQL

Those details are left to each tool or library. As a result, Amazon Neptune, AllegroGraph, GraphDB, Stardog, Jena, and others behave differently in how named graphs are created, managed, and listed, even though they all support the same SPARQL surface language.

For design modeling, the key takeaway is subgraphs are conceptually standardized (via RDF Datasets and SPARQL), but the mechanics of creating and managing them are implementation specific. This raises the question of how to make decisions about partitioning system knowledge into subgraphs. The next sections cover the mechanisms for doing so; after that, we turn to the design principles that help determine when those mechanisms are appropriate.

How to Create Subgraphs

Using subgraphs requires us to use a different formalism than a triple. The formalism required is called an *N-Quad*. An N-Quad extends an RDF triple with a graph identifier, enabling statements to be organized into named graphs for context, provenance, and data management.

The details vary by language and client library, but the pattern is the same. You specify a subject, predicate, and object. Then you also specify which subgraph (one or more) that triple belongs to. In many systems, creating a subgraph is as simple as inserting the first triple into that graph IRI. In others, you may explicitly issue a SPARQL Update like

```
CREATE GRAPH <https://example.com/graphs/Raw_Graph>
```

before inserting. Either way, the design concept is identical: a subgraph is the set of all triples that share the same graph identifier.

This works equally well for

- An OWL ontology graph
- An RDF data graph
- A pipeline staging graph

Moving Triples Between Subgraphs

Because subgraphs are sets of triples, moving data between them is conceptually straightforward. SPARQL 1.1 Update provides both high-level and low-level ways to do this. The SPARQL queries that follow illustrate the basic kinds of subgraph manipulation you can do. These can all be found in https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_8/Listings_8-1_to_8-21.sparql.

If you want to move all triples, you can use the MOVE operation:

```
MOVE GRAPH <https://example.com/graphs/Raw_Graph>
TO   GRAPH <https://example.com/graphs/Cleaned_Graph>
```

This operation is defined in the SPARQL 1.1 Update specification. Under the hood, the store will copy triples and then clear the source graph.

More often, we want to move only a subset of triples. For example, only validated measurements of type `Climate_Observation`. In that case, we can use an `INSERT... WHERE / DELETE ... WHERE` pattern (Listing 8-1).

```
INSERT {GRAPH <https://example.com/graphs/Validated_Graph>
{?observation a :Climate_Observation ;
:has_measurement_value ?value. }}
WHERE {GRAPH <https://example.com/graphs/Raw_Graph>
{?observation a :Climate_Observation ;
                :has_measurement_value ?value.
    ?observation :validation_status "valid". }};
DELETE {GRAPH <https://example.com/graphs/Raw_Graph>
    {?observation a :Climate_Observation ;
                :has_measurement_value ?value ;
                :validation_status "valid".}}
WHERE {GRAPH <https://example.com/graphs/Raw_Graph>
    {?observation a :Climate_Observation ;
                :has_measurement_value ?value ;
                :validation_status "valid".}}
```

Listing 8-1 SPARQL Update to Move Triples from One Subgraph to Another

Here, we are

- Copying only the validated observations into `Validated_Graph`
- Removing those same validated triples from `Raw_Graph`

At the design level, this supports a clean pipeline:

```
Raw_Graph → Cleaned_Graph → Validated_Graph → Published_Graph
```

mirroring the DAG view discussed below. Clients in Python, Java, etc., often provide helper methods that abstract this pattern, but SPARQL Update is the portable core.

Subgraphs As Encapsulation Boundaries

OWL does not support encapsulation. Encapsulation is the definition of boundaries for the knowledge graph that define the scope for entities (e.g., the division between public vs. internal entities). This is appropriate because OWL models the conceptual structure of a domain, while visibility and interface boundaries belong in the design layer. Encapsulation, a critical consideration in designing for robustness, is accomplished through the other, complementary tools.

Providing these mechanisms is one of the ways that SHACL, named graphs, and SPARQL access rules supplement OWL.

Subgraphs are a key design tool. They are used in many ways. For example, they can separate TBox axioms from ABox assertions, triples inferred by the reasoner vs. those defined by the user, and entities that have different levels of security and visibility.

Subgraphs provide a natural unit for governance, access control, validation, and monitoring. These can all be applied at the graph level, which is often more convenient than at the level of individual triples. Because each subgraph represents a cohesive domain, they function as the semantic equivalent of bounded contexts in Domain-Driven Design (which we'll discuss in **Chapter 13**).

The next two subsections drop back down to some mechanics for dealing with subgraphs: querying them and manipulating them. After that, we'll move on from encapsulation to data ingestion.

Querying Subgraphs in SPARQL

SPARQL uses the GRAPH keyword to target a specific subgraph inside a dataset. For example, if all validated climate observations live in `Validated_Graph`, we might write the query shown in Listing 8-2.

```
PREFIX : <https://www.michaeldebellis.com/climate/>
SELECT ?observation ?value
WHERE {GRAPH <https://example.com/graphs/Validated_Graph>
    {?observation a :Climate_Observation ;
                  :has_measurement_value ?value. }}
```

Listing 8-2 SPARQL Query Confined to a Specific Subgraph

We can also join across graphs, for example, using an ontology from `Ontology_Graph` and data from `Data_Graph` in one query. Some triple stores materialize reasoning results into separate graphs (e.g., `Inferred_Graph`), and SPARQL makes it straightforward to query only asserted data, only inferred data, or both.

For a "cheat sheet" on all the basic operations to manipulate graphs in SPARQL, see https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_8/SPARQL_graph_patterns_cheat_sheet.ru.

Data Ingestion As a First-Class Design Activity

Expecting data to be dirty is a norm.
—Dehghani, Zhamak. Data Mesh: Delivering Data-Driven Value at Scale (p. 265). O'Reilly Media. Kindle Edition.

One of the most critical activities in design modeling is working with representative real-world data and defining ingestion pipelines. As with conceptual modeling, real data clarifies design alternatives. However, in conceptual modeling, we weren't concerned with issues such as how often real data fails to conform to datatype constraints nor with requirements about data quantity and response time. In design, these types of requirements are typically most important.

Across both large enterprises and small startups, I have consistently found that teams assert strong guarantees about their data such as "this column always contains a valid floating-point number" or "this field always contains an `xsd:dateTime`" that often don't survive contact with reality. In Semantic Knowledge Graphs, these assumptions matter because they interact directly with OWL's formal semantics. A single invalid literal can render an OWL reasoner ineffective until the issue is resolved.

A common example of this problem is the presence of placeholder values such as "N/A" or "None" in fields that are modeled with datatypes such as `xsd:decimal` or `xsd:dateTime`. These cases are so common that I now assume they will occur in any nontrivial dataset.

This immediately raises a design decision: when and where should data be transformed to conform to the ontology? Before we answer that, we need to clarify some terminology.

It is useful to distinguish between *data ingestion* and *data pipelines*, even though the terms are often used interchangeably. Data ingestion refers to the act of bringing data into the Semantic Knowledge Graph: assigning identifiers, mapping records to ontology classes and properties, and recording provenance about when and how the data entered the system. A data pipeline is broader: it is an overarching orchestrated process that subsumes ingestion, validation, enrichment, movement between subgraphs, and publication.

This distinction is especially important in Semantic Knowledge Graphs, where ingestion events are often modeled explicitly (in the example in this chapter as `prov:Activity`), while pipelines are represented implicitly as workflows or directed acyclic graphs (DAGs) that manage how data evolves over time. In the StreamForge example below, the pipeline is orchestrated using the Prefect Python library.

From a design perspective, data ingestion typically occurs in one of three modes: batch ingestion, request-based ingestion via APIs, and streaming ingestion from event platforms. Batch ingestion is common

for proof-of-concept prototypes, periodic snapshots, and large datasets. API-based ingestion supports near-real-time updates driven by operational systems. Streaming ingestion treats the knowledge graph as a consumer in an event-driven architecture.

While these modes differ operationally, they raise similar design questions for Semantic Knowledge Graphs. For example, how ingestion events are identified, how provenance is captured, how failures are handled, and how data is staged and validated. For example, batch ingestion often models provenance at the file or dataset level, while streaming ingestion may require modeling each event or window of events as a distinct activity. These choices affect graph structure, query patterns, and system performance, and therefore belong squarely in the design model.

Introducing the StreamForge Domain

This chapter introduces a hypothetical company called StreamForge, which will be used throughout the remainder of the book to illustrate practical concepts with realistic examples. Earlier chapters relied exclusively on the People Ontology, which was a deliberately minimal example to explain foundational Semantic Web concepts. While pedagogically effective, those examples were intentionally abstract and do not reflect the kinds of data, systems, and organizational boundaries encountered in real-world enterprise environments. The StreamForge domain is introduced here to provide a hypothetical Internet business.

StreamForge represents a video content streaming platform that aggregates and analyzes engagement data about videos. The domain includes familiar concepts such as videos, channels, categories, regions, rankings, and time-based snapshots, making it suitable for illustrating data pipelines, provenance, and lifecycle concerns without requiring specialized domain knowledge.

Relationship to Dehghani's Daff Example

The StreamForge example is inspired in part by the hypothetical company Daff, introduced by Zhamak Dehghani in (Dehghani, 2022). Dehghani's Daff example provides a rich and compelling narrative for explaining data mesh principles, domain ownership, and the cultural shift required to treat data as a first-class product in the streaming audio domain.

StreamForge serves a similar purpose: it is a deliberately simplified company created specifically to support examples that have similar complexity to real industry domains. In later chapters, the StreamForge domain will be revisited when introducing data products, bounded contexts, and Domain-Driven Design.

Data Ingestion Pipeline Example: StreamForge

In a Semantic Knowledge Graph, ingestion is the workflow where raw data is mapped into a shared semantic structure, where provenance is generated, and where the boundaries between schema (TBox), data (ABox), and metadata (provenance) become visible and intentional.

Even a minimal ingestion pipeline illustrates several crucial ideas:

- Raw records become individuals in the ABox.
- Metadata about the ingestion run itself becomes part of the graph.
- The ontology (TBox) remains separate but participates in the interpretation of the data.
- Subgraphs (named graphs) enforce boundaries between these layers.
- Ingestion is repeatable and produces a growing set of events, not a one-time import.

To make these ideas concrete, we walk through a complete, small-scale ingestion pipeline. The data pipeline is illustrated as a DAG in Figure 8-1. This section presents a concrete, end-to-end data ingestion example that illustrates how operational data can be transformed into a Semantic Knowledge Graph while preserving explicit provenance about the ingestion process itself. The Python Prefect code as well as the example CSV, Turtle, and other files can be found at https://github.com/mdebellis/SemanticKG-Design/tree/main/chapter_8/data_ingestion.

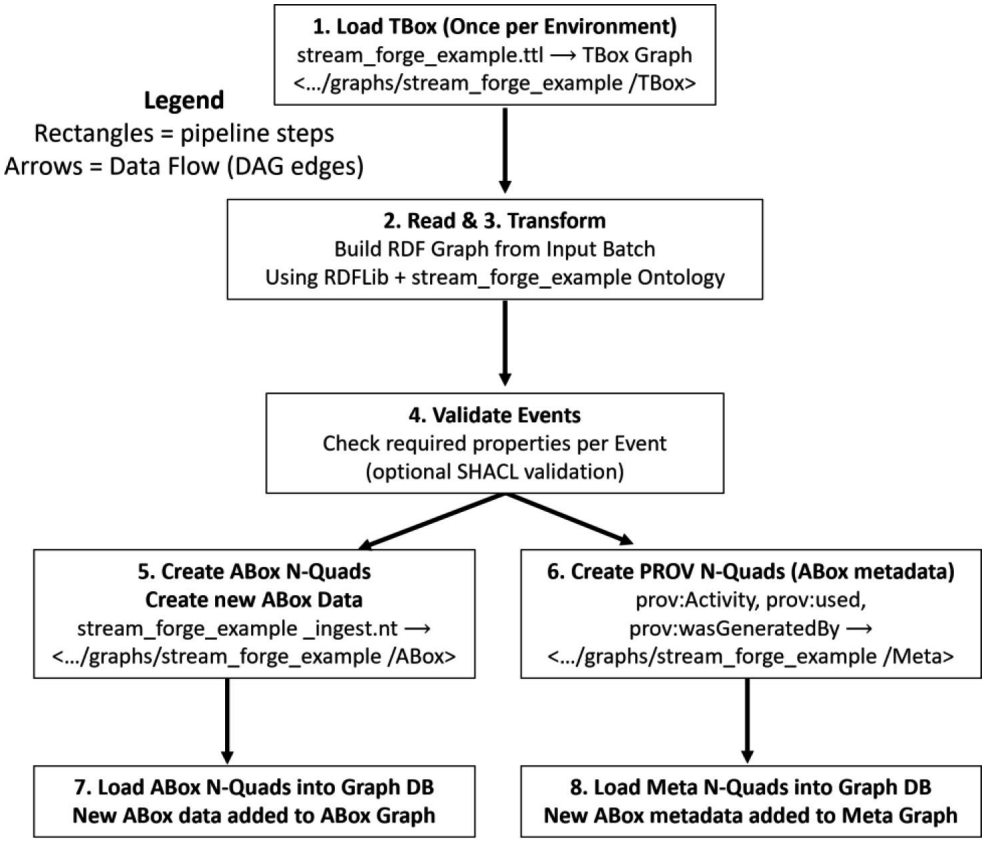


Figure 8-1 Data pipeline modeled as a directed acyclic graph (DAG)

Overview of the Example

The definition of the data pipeline using the Python tool Prefect is in the file: `stream_forge_example_prefect.py`. The ingestion pipeline performs the following steps (see Figure 8-1):

1. *Load TBox*: This is done once to set up the environment. This loads the OWL file with the TBox definitions of classes and properties. This defines the knowledge graph that the new data will be loaded into. For this small example, this is the file: `ingestion_example.ttl`.
2. *Read New Input*: In the diagram, this and the next step are combined in the activity Read and Transform. A new run of the pipeline is triggered by a file being added to the `input_data` directory. In this example, the CSV file `stream_forge_example_trending_snapshot.csv` is added to the directory and triggers the pipeline. This file has recent data about trending videos. In the repository, this file is in the `processed_data` subdirectory because we are showing the state after the pipeline has completed ingesting the latest data.
3. *Transform New Input*: Parse the CSV file in that directory and construct RDF triples representing domain entities.
4. *Validate Events*: Validate the new triples against the SHACL file that defines data integrity constraints for trending videos: `trending_snapshot_validation.ttl`
5. *Create ABox N-Quads*: We use the N-Quads format for the ingested video rather than Turtle because that format allows us to include named graphs with each RDF triple. This and the next activity create the triples in memory (but not yet saved to the knowledge graph).

6. *Create PROV-O Metadata*: In addition to the actual ABox data loaded, we also generate and record provenance data describing the ingestion run using the PROV-O ontology.
7. *Load ABox N-Quads*: Load the new data from Activity 5 into the graph database and the appropriate subgraph.
8. *Load Metadata N-Quads*: Load the metadata from Activity 6 into the graph database and the appropriate named graph.

Input Data

Pipelines can be set to run on scheduled intervals or be triggered by events. In this example, the pipeline is triggered by a new file being placed in the `input_data` directory. The input is a CSV file representing a snapshot of trending video data captured at a specific point in time. A trending video is a video that is currently receiving unusually high attention relative to other videos on the platform. Each row in the file corresponds to a single video appearing in the trending list of the StreamForge site at the moment the snapshot was taken.

Typical fields include

- Video identifier
- Video title
- Channel name
- Category
- Region
- Rank
- Likes (24h)
- Comments (24h)
- Snapshot timestamp

The CSV file is treated as immutable input. Once processed, it is automatically moved out of the input directory to prevent accidental re-ingestion.

Ingestion Run As a First-Class Concept

Each pipeline execution is modeled explicitly as an ingestion run, identified by a timestamp-based IRI. This ingestion run serves as the central node connecting

- The input dataset
- The pipeline software agent
- The generated domain records
- The time at which the ingestion occurred

Treating the ingestion run as a first-class entity makes provenance explicit and queryable, rather than implicit in logs or filenames.

Constructing the Domain Graph (ABox)

For each row in the CSV file, the pipeline creates a Trending Snapshot Record representing the appearance of a particular video in the trending list at a specific snapshot time.

Each snapshot record is linked to

- A Video entity
- A Channel entity
- Literal values such as rank, category, region, likes, and comments

Videos and channels are represented as reusable entities, allowing future ingestions to refer to the same videos or channels across multiple snapshots. The resulting ABox graph captures what was observed in the trending data at the time of ingestion, independent of how or when it was processed.

Capturing Provenance (Meta Graph)

In addition to the domain data, the pipeline generates explicit provenance using the W3C PROV-O model. At the OWL level, these are also ABox data: instances and property values. However, we save them in a different named graph to differentiate the new domain data and the metadata about the domain data.

The provenance graph records

- The ingestion run as a `prov:Activity`
- The pipeline software as a `prov:SoftwareAgent`
- The input CSV file as a `prov:Entity`

The relationships between them:

- The ingestion run `prov:used` the input file.
- The snapshot records `prov:wasGeneratedBy` the ingestion run.
- The ingestion run `prov:wasAssociatedWith` the pipeline agent.
- The run has explicit start and end times.

This provenance information answers questions such as:

- Which ingestion run produced a given snapshot record?
- Which input file was used?
- When did the ingestion occur?
- Which software agent performed the ingestion?

Output Files and Graph Separation

The pipeline writes its output into a timestamped directory, producing two RDF files:

- An ABox file containing domain data
- A Meta file containing provenance data

These files are then loaded into AllegroGraph using three named graphs, corresponding to the TBox (schema metadata), ABox (actual data), and Meta (metadata related to the `prov:Activity` such as provenance data):

```
<https://www.michaeldebellis.com/stream_forge_example/graphs/TBox>
<https://www.michaeldebellis.com/stream_forge_example/graphs/ABox>
<https://www.michaeldebellis.com/stream_forge_example/graphs/Meta>
```

Visualizing the Resulting Knowledge Graph

Once loaded into AllegroGraph, the data can be explored using Gruff. The visualization in Figure 8-2 shows

- The ingestion pipeline at the center of the graph
- Trending snapshot records generated by that run with `prov:wasGeneratedBy` links to the ingestion pipeline
- Example data such as videos, channels, and view count connected to each snapshot
- The input CSV file used by the ingestion
- Temporal metadata associated with the run

This makes it possible to visually trace how domain data was produced and to distinguish between observed content and process metadata. The diagram below shows some of the data and metadata added to the knowledge graph.

Why This Pattern Matters

Although the example is intentionally simple, it demonstrates several important principles:

- Data ingestion is modeled, not just executed.
- Provenance is explicit, queryable, and represented using a widely adopted W3C standard (Prov W3C Working Group, 2013).
- Operational metadata lives alongside domain data rather than in external logs.
- The same pattern can support batch, streaming, or scheduled ingestion workflows.

Most importantly, the example shows how a Semantic Knowledge Graph can unify data, metadata, and process context in a single, coherent model.

The preceding sections described ingestion and transformation concerns in a systematic way. We now turn to a concrete example that illustrates how these concepts are modeled and recorded in our example Semantic Knowledge Graph pipeline. This model is illustrated in Figure 8-2.

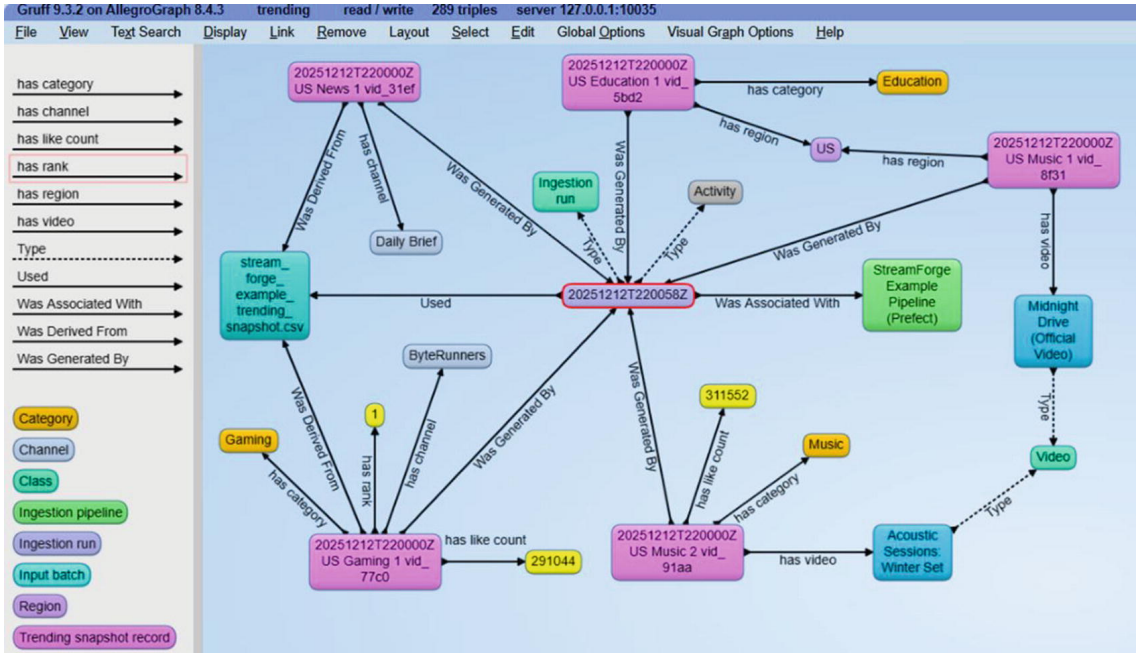


Figure 8-2 Data and metadata added to graph by data pipeline

At the center of the graph is an `Ingestion_Run`, modeled as a `prov:Activity`. This node represents one concrete execution of the ingestion pipeline, corresponding to a single Prefect flow run. The ingestion run is associated with an Ingestion Prefect Pipeline, modeled as a `prov:SoftwareAgent` (see the green node to the right linked to the `Ingestion_Run` by the `prov:wasAssociatedWith` link), making the automatic process that produced the new data explicit, and part of the Meta graph associated with the data.

The ingestion run uses an Input Batch CSV file, modeled as a `prov:Entity`. See the trending snapshot node in the middle left of Figure 8-2. This input batch represents the external data source processed during the run. Treating the input batch as a first-class entity allows the knowledge graph to retain a durable reference to the source data that gave rise to the ingested data.

Trending video records are shown as domain entities generated by this ingestion run. Each record is explicitly linked to the ingestion run using the `prov:wasGeneratedBy` property. The records are also linked back to the input batch using `prov:wasDerivedFrom`, making it possible to trace any individual fact to the specific snapshot from which it originated.

From each trending record, the graph connects to related domain entities such as videos, channels, categories, and regions using domain-level properties (e.g., linking a trending record to the video it refers to or the region in which it was observed). These links represent the business meaning of the data, while the

provenance links capture its lifecycle and origin. Showing all these links would result in a fairly busy graph, so Figure 8-2 has a few representative values from each trending record such as the category, region, and video.

Timestamps recording when the pipeline started and completed are attached to the ingestion run. Together with the associations to the pipeline software and input batch, this provides a complete, queryable audit trail answering questions such as:

- Which pipeline produced this data?
- When was it ingested?
- What source data was used?
- Which trending records were generated together as part of the same run?

This example demonstrates a key design principle of Semantic Knowledge Graphs in production systems: domain facts and pipeline metadata coexist in the same graph while remaining conceptually distinct. By modeling provenance explicitly rather than relying solely on external logs or orchestration tools, the knowledge graph becomes self-describing with respect to its own construction, enabling transparency, traceability, and governance without sacrificing usability or clarity.

ABox queries confirm that the trending video records were ingested correctly. More interestingly, queries that combine the ABox data and provenance metadata highlight the relationship between ingestion processes and domain knowledge, making it possible to reason about how and when the graph acquired its current state.

If ingestion exposes mismatches between design assumptions and real-world data, the need arises to make those assumptions explicit and enforceable. This is where SHACL becomes critical.

Making Design Assumptions Explicit

The design considerations discussed above raise an important but subtle issue. Many of the assumptions that shape data ingestion are already present in the ontology, even when they are not made explicit as validation rules. Declaring a property with a datatype range, or modeling a property as functional, reflects a clear expectation about what valid data should look like. What is often missing is a systematic way to check whether incoming data conforms to those expectations. This is where SHACL comes in.

First, though, some background. At this stage, it is useful to revisit the distinction between logical modeling and constraint validation. OWL ontologies support reasoning under the Open World Assumption (OWA), allowing new knowledge to be inferred from what is stated. They are not designed to flag violations when data fails to meet an intended constraint. They are used to reason about the ontology. For example, if the domain of `has_view_count` is the Video class and an object has a value for `has_view_count`, then the reasoner will infer that the object is an instance of `Video`. The design of OWL is most appropriate for conceptual models of large, heterogeneous, distributed data. The Open World Assumption (OWA), which assumes there may be data that is not in the system, works well for such models.

However, when doing data integrity validation, the OWA is not appropriate. Recall that an OWL axiom that requires each `Employee` to have an `employee_id` will not trigger an error if an `Employee` doesn't have such an id (the OWA means the id may be in a database that has not yet been imported into the knowledge graph). However, for doing data validation, saying "there is no required value… but it may be out there somewhere" is not appropriate.

SHACL, by contrast, is explicitly concerned with validation. It uses the more conventional Closed World Assumption (CWA) to check whether the data conforms to stated constraints, without attempting to infer new facts. The CWA assumes all data is in the system and consequently triggers warnings when required data isn't present.

As ingestion exposes mismatches between design assumptions and real-world data, the need arises to make those assumptions explicit and enforceable. This is where the distinction between OWL and SHACL becomes critical.

SHACL As a Derived Design Artifact

It is often productive to view SHACL constraints as derived artifacts from the OWL axioms in the conceptual model. SHACL is not a parallel model; it is a mechanically derivable view of design intent. Many commonly needed constraints correspond directly to OWL modeling choices. For example:

- A datatype property with an `rdfs:range` implies an expected datatype.
- A functional property implies that at most one value should be present.

Because both OWL ontologies and SHACL shapes are represented in RDF, these assumptions can be surfaced automatically by transforming one RDF graph into another.

SPARQL plays a central role in this process. As a pattern-matching and graph construction language, it provides a uniform mechanism for querying ontologies and generating corresponding validation artifacts. This makes it possible to derive SHACL constraints automatically, without duplicating design decisions or maintaining parallel models by hand.

Once design assumptions are expressed as validation rules, the next step is to operationalize them. These architectural constraints also influence how data can be queried and accessed. SPARQL provides a uniform mechanism for doing so, enabling both *the extraction of design patterns from ontologie*s and the *generation of corresponding SHACL constraints*.

Using SPARQL to Operationalize Design (OWL → SHACL)

The following examples demonstrate this idea using some simple SPARQL `CONSTRUCT` examples. Their purpose is to illustrate how common OWL modeling patterns can be made explicit as SHACL constraints in a way that is easy to understand and reason about. In the next chapter, we will show a common coding pattern where we take SPARQL queries and updates and wrap them in a function.

Datatype Ranges As Validation Constraints

When an ontology declares a datatype property with an `rdfs:range` such as `xsd:decimal` or `xsd:dateTime`, it encodes a clear design assumption about permissible values. In OWL, this assumption influences reasoning but does not prevent invalid literals from being asserted.

A corresponding SHACL constraint makes the assumption explicit by stating that all values for the property must conform to the declared datatype. The first example shows how a SPARQL query can identify datatype properties with XSD ranges and generate SHACL property shapes with `sh:datatype` constraints. Doing so directly operationalizes a design decision that is already present in the ontology.

This example uses the People Ontology to illustrate deriving SHACL property constraints from OWL datatype properties which declare an `rdfs:range` using an XSD datatype. The query constructs one SHACL `PropertyShape` per such property and attaches it to a simple container `NodeShape`.

This is one of the most common transformations when doing design with real-world data. It is often the case that data needs to be ingested into the ontology as soon as possible. With a single invalid datatype, the ontology would no longer be consistent, and the reasoner would be unusable until the invalid datatype is repaired (by which time you may have ingested more bad data). In fact, this is one of the use cases driving demand for SHACL, which wasn't initially part of the Semantic Web stack. Listing 8-3 shows a SPARQL transformation that generates SHACL datatype constraints from OWL datatype-property ranges.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX sh:   <http://www.w3.org/ns/shacl#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT {<https://www.michaeldebellis.com/people/shapes/generated>
    a sh:NodeShape;
    sh:property ?ps.
```

```
  ?ps a sh:PropertyShape ;
      sh:path ?p ;
      sh:datatype ?dt ;
      sh:name ?label.}
WHERE {?p a owl:DatatypeProperty ;
     rdfs:range ?dt.
  FILTER(STRSTARTS(STR(?dt), STR(xsd:)))
  OPTIONAL { ?p rdfs:label ?label }
  # Use local name after your People namespace to build readable shape IRIs
  BIND( STRAFTER(STR(?p), "https://michaeldebellis.com/people/") AS ?local )
  # Fallback if STRAFTER() didn't match (keeps query robust)
  BIND( IF(BOUND(?local) && STRLEN(?local) > 0, ?local, ENCODE_FOR_URI(STR(?
p))) AS ?id )
BIND(IRI(CONCAT("https://www.michaeldebellis.com/people/shapes/ps/datatype/",
?id))
    AS ?ps )}
```

Listing 8-3 SPARQL Transformation to Generate SHACL Constraints from OWL Axioms

This is a simple example. The function https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_8/owl_to_shacl.py is an example of a more sophisticated function that can also remove the datatype axiom from the OWL model. This is a common design activity: define datatypes first in the conceptual model, then if needed due to bad data, migrate those axioms to SHACL constraints so that the bad data doesn't make the ontology model inconsistent.

This is somewhat different from the way OWL and SHACL are often described. Other books treat OWL and SHACL as essentially different unrelated languages. The common mantra is *OWL is for reasoning and SHACL for constraints*. However, I've found that in large deployed systems, this approach of understanding the conceptual model with OWL then transforming some axioms (especially datatype property range axioms) into SHACL based on real-world data is often a productive way to leverage the two.

Functional Properties and Cardinality Expectations

A similar pattern applies to functional properties. Declaring a property as functional expresses the intent that an individual should have at most one value for that property.

By generating a SHACL constraint with a `sh:maxCount 1` restriction, this same design intent can be enforced as a validation rule. Listing 8-4 shows how functional properties can be transformed into cardinality constraints using a simple SPARQL query. As before, the goal is not to alter the ontology's semantics, but to make an implicit design assumption explicit at validation time.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX sh:   <http://www.w3.org/ns/shacl#>
CONSTRUCT {<https://www.michaeldebellis.com/people/shapes/generated>
    a sh:NodeShape ;
    sh:property ?ps.
  ?ps a sh:PropertyShape ;
      sh:path ?p ;
      sh:maxCount 1 ;
      sh:name ?label.}
WHERE {?p a owl:FunctionalProperty.
  OPTIONAL { ?p rdfs:label ?label }
  # Use local name after your People namespace to build readable shape IRIs
```

```
  BIND( STRAFTER(STR(?p), "https://michaeldebellis.com/people/") AS ?local )
  # Fallback if STRAFTER() didn't match
  BIND( IF(BOUND(?local) && STRLEN(?local) > 0, ?local, ENCODE_FOR_URI(STR(?
p))) AS ?id )
  BIND(
IRI(CONCAT("https://www.michaeldebellis.com/people/shapes/ps/functional/", ?
id)) AS ?ps )}
```

Listing 8-4 SPARQL to Generate SHACL Cardinality Constraints from OWL Axioms

Design Implications

These examples illustrate a broader design principle. In Semantic Knowledge Graphs, design models are not merely documentation artifacts. Because they are expressed in RDF, they can be queried, transformed, and used to generate operational components such as validation rules, governance checks, or ingestion controls.

Treating SHACL constraints as derived products of ontology design helps preserve a clear separation of concerns. In the implementation chapters that follow, we will examine how this approach can be applied more systematically by wrapping SPARQL updates with Python functions.

Having seen how SPARQL can be used to automate creation of SHACL from OWL design, we next address how to write efficient SPARQL queries.

Designing Efficient SPARQL Queries

SPARQL is more than a query language. It can also be used to load and most importantly transform data. One of the most elegant design decisions of the Semantic Web is that all the languages ultimately come down to RDF triples. Thus, pattern matching in SPARQL, where a pattern can match any or all of a triple, enables using SPARQL to automate many types of refactoring. An important example is coercing values to appropriate data types and transforming models from one language (e.g., OWL) to another (e.g., SHACL).

The power of SPARQL to generate all permutations of a triple pattern also allows unintentionally writing perfectly reasonable-looking SPARQL queries that create combinatorial explosions. This section will illustrate some of the most common errors that novice SPARQL users make and will present some simple habits for utilizing the power of SPARQL that avoid combinatorial explosion.

SPARQL Anti-pattern 1: The Cartesian Product

For example, suppose we want to discover what information is stored in the People Ontology about Albert Einstein. Even if we do not know which properties are used, we can write a simple query that retrieves all triples where Einstein appears as the subject (Listing 8-5).

```
PREFIX :  <https://michaeldebellis.com/people/>
SELECT *
WHERE {:Albert_Einstein ?p ?o.}
```

Listing 8-5 A Simple Query to Return All Triples with Einstein As Subject

This query returns all property–value pairs associated with Einstein. That is, all the properties that are values associated with Einstein such as `has_colleague`. Queries like this are extremely useful during early exploration, debugging, and learning an unfamiliar dataset. They also illustrate a key strength of RDF: the ability to discover structure dynamically rather than relying on predefined schemas.

Why `DISTINCT` Is Often a Good Idea

When using broad queries such as `SELECT *`, it is usually a good idea to add the `DISTINCT` keyword (Listing 8-6).

```
# DISTINCT removes duplicates. As with all queries
# remember to include the PREFIX from Listing 8-5
```

```
SELECT DISTINCT *
WHERE { :Albert_Einstein ?p ?o.}
```

Listing 8-6 Using DISTINCT to Avoid Duplicate Triples in Your Results

Although RDF graphs themselves should not contain duplicate triples, it is possible for queries, code, or loading different datasets to result in duplicate triples. More importantly, even when there are no actual duplicate triples, SPARQL query results can contain duplicate triples. This can happen for several reasons, including inference, joins against multivalued properties, or query patterns that match the same triple through multiple paths. Using `DISTINCT` helps ensure that the results are easier to read and reason about, especially when a query is intended for exploration rather than analytics.

At first glance, it may seem surprising that a SPARQL query can return duplicate results. After all, an RDF graph is defined as a set of triples, and sets should not contain duplicates. The explanation lies in the fact that SPARQL does not operate directly on triples in isolation. Instead, it operates on *pattern matches* and produces *solution mappings*.

Conceptually, a SPARQL query proceeds in two steps. First, the query engine finds all ways in which the triple patterns in the WHERE clause can be matched against the graph. Each successful match produces a binding of variables to RDF terms that is called a "solution mapping." Second, the query engine projects these solution mappings into a result set.

Importantly, different solution mappings can project to the same visible result. This can happen even when the underlying RDF triples are unique.

For example, consider the simple query for all triples that have Einstein as the subject in Listing 8-7.

```
PREFIX :  <https://michaeldebellis.com/people/>
SELECT ?p ?o
WHERE {:Albert_Einstein ?p ?o.}
```

Listing 8-7 Finding Triples with Einstein As Subject

If the graph contains inference rules, property hierarchies, or multiple ways of deriving the same relationship, the same (`?p, ?o`) pair may be matched through different logical paths. Each path produces a distinct solution mapping, even though the projected result appears identical to the user.

For example, in the People Ontology, `has_father` is a sub-property of `has_family_relationship` and a sub-property of `has_parent`. Since the graph asserts `Albert_Einstein has_father Hermann_Einstein`, then the inferred triple `Albert_Einstein has_family_relationship Hermann_Einstein` can be derived through two paths: `has_father` and `has_parent`.

Similarly, joins against other patterns such as optional clauses, multivalued properties, or inferred type relationships can multiply solution mappings without introducing any new triples into the graph. From the perspective of the query engine, these are distinct matches, even if they look redundant in the final output.

By default, SPARQL preserves all such matches. The result of a SELECT query is therefore a multiset (sometimes called a "bag" because unlike a set, it can contain the same element multiple times) of solution mappings rather than a mathematical set. This is useful because it allows you to use SPARQL to test if your graph has redundant data. However, it also means that duplicate-looking rows are often a natural consequence of query evaluation.

Using `DISTINCT` instructs the query engine to remove duplicate rows after the solution mappings have been generated and projected. It does not change how the query is matched against the graph, nor does it improve the efficiency of poorly structured queries. Instead, it ensures that the final results are easier to interpret when duplicates are not meaningful for the task at hand.

Understanding the distinction between RDF triples and SPARQL solution mappings is essential for writing clear and predictable queries. It also explains why `DISTINCT` is commonly used in exploratory queries, even when working with clean, well-designed RDF data.

A natural next step is to retrieve not only the triples where Einstein is the subject, but also the triples where other resources point to Einstein. Conceptually, this corresponds to retrieving the immediate

"neighborhood" of a node in the graph. Listing 8-8 shows what a first (but ill-advised) attempt might look like.

```
# Anti-Pattern! This is a BAD query! Don't do this!
PREFIX :  <https://michaeldebellis.com/people/>
SELECT DISTINCT *
WHERE {?s ?p1 :Albert_Einstein.
       :Albert_Einstein ?p2 ?o.}
```

Listing 8-8 A BAD Example That Will Generate Unneeded Extra Solutions. Do NOT Do This!

At first glance, this query seems reasonable. However, it is a classic example of a query that can cause serious performance problems.

SPARQL matches all triple patterns in the `WHERE` clause simultaneously. When two or more patterns do not share any variables, the query engine forms the Cartesian product of all matching solutions. In this example, the two patterns are independent of one another: one matches incoming triples to Einstein, and the other matches outgoing triples from Einstein.

If there are M triples that point to Einstein and N triples that originate from Einstein, the query engine will generate M × N intermediate solution mappings. Although many of these intermediate results are discarded before producing the final output, the cost of having unnecessarily generated them can be very high. Especially when using real data, queries like this can soon add thousands and even millions of useless solution mappings. Since these mappings aren't real triples, they won't be saved when you save your graph. But they ***will*** take up space in your graph database and make your query take longer than needed.

Many SPARQL tools include query-planner warnings that alert you when a query contains a join that will produce a cross product. A typical warning indicates that the query includes a join where the variables on the left-hand side of the join do not overlap with the variables on the right-hand side. In other words, the query engine has detected that two independent sets of solution mappings will cause a combinatorial explosion for graphs that have even small amounts of data.

Such warnings are often phrased in terms of unshared variables or independent join operands and may explicitly state that the join will result in a cross product. While the exact wording varies by tool, the intent is the same: the query is likely to generate a large number of useless intermediate results.

When you encounter a warning of this kind, it is a strong signal that the query structure should be reviewed. In many cases, the issue can be resolved by rewriting the query to keep independent patterns separate using `UNION` as shown below. This kind of problem is especially common in queries that "look simple" but unintentionally combine independent patterns.

Using UNION to Avoid Computational Complexity Problems

The correct way to retrieve both incoming and outgoing triples is to keep the two patterns independent using `UNION`. The `UNION` keyword in SPARQL is the same as union in set theory, as discussed in **Chapter 3**. It combines `Result1` and `Result2` into a single result with all the elements that are in either `Result1` or `Result2` but without any duplicates.

Each branch of a `UNION` produces a set of solution mappings (bindings of variables to RDF terms). The result of the `UNION` is the combination of these solution mappings from both branches into a set. Listing 8-9 shows an example.

```
SELECT ?s ?p ?o
WHERE {
  {:Albert_Einstein ?p ?o.
    BIND(:Albert_Einstein AS ?s)  }
  UNION
  {?s ?p :Albert_Einstein.
    BIND(:Albert_Einstein AS ?o) }}
```

Listing 8-9 Using UNION to Create a More Efficient Query

The UNION version treats them as separate alternatives, so the number of solution mappings grows additively. Rather than M × N, this version will return (at most) M + N solution mappings.

In this example (and this is often the case), using `CONSTRUCT` to return a subgraph rather than a table will be the better alternative. A `SELECT` query returns tables of variable bindings. However, conceptually many exploratory queries are asking for a subgraph. In such cases, a `CONSTRUCT` query is often a better choice (Listing 8-10).

```
CONSTRUCT {:Albert_Einstein ?p ?o.}
WHERE {:Albert_Einstein ?p ?o.}
```

Listing 8-10 Using CONSTRUCT Returns Triples Rather Than Tables

This query returns an RDF graph rather than a table. The result can be visualized directly, merged into another graph, or processed by downstream tools without relying on client-specific conventions for interpreting tabular results as triples. When the intent of a query is to retrieve part of a graph, using `CONSTRUCT` makes that intent explicit and avoids ambiguity. Note that a `CONSTRUCT` query does not add the triples to the graph. It simply creates new triples and then it is up to the user what to do with them. This can be a bit confusing because the only way to add new triples using SPARQL in Protégé is with the `CONSTRUCT` keyword. However, that is a hack required since the SPARQL engines in Protégé don't implement `INSERT` or `DELETE`. Normally, to add triples, you use `INSERT`.

Using `CONSTRUCT` also makes the intent of a query clearer. A `SELECT` query answers questions such as "Which values match these variables?" In contrast, a `CONSTRUCT` query answers a different kind of question: "Which part of the graph do I want to retrieve?" When the goal is to extract or create a subgraph such as all information about a particular resource, `CONSTRUCT` expresses that intent directly.

Returning triples preserves the semantics of the data. RDF graphs are sets of triples, and duplicate triples are naturally eliminated. In contrast, `SELECT` queries return multisets of solution mappings, which can include duplicate rows even when the underlying data is clean. When the goal is to work with knowledge rather than perform analytics, operating directly on triples is usually the simpler and safer choice.

For these reasons, `SELECT` is best suited to reporting, analytics, and debugging, while `CONSTRUCT` is the better choice for design, exploration, and integration tasks. Listing 8-11 uses CONSTRUCT to retrieve both outgoing and incoming relationships for Einstein in a way that does not create a combinatorial explosion.

```
CONSTRUCT { :Albert_Einstein ?p ?o.
            ?s ?p2 :Albert_Einstein.}
WHERE {
  { :Albert_Einstein ?p ?o }
  UNION
  { ?s ?p2 :Albert_Einstein } }
```

Listing 8-11 Using CONSTRUCT Returns Triples Rather Than Tables

In both cases, each pattern is evaluated independently, and the results are combined without creating a Cartesian product. This approach scales much better and more accurately reflects the conceptual intent of the query.

Additional SPARQL Anti-patterns to Be Aware Of

Writing efficient SPARQL queries involves far more than can be covered in this chapter. While the SPARQL specification precisely defines query semantics, it intentionally leaves many aspects of query planning and optimization to individual implementations. As a result, guidance on efficient query design is scattered

across standards documents, implementation notes, and practitioner experience rather than consolidated in a single reference. I focused on the Cartesian product anti-pattern because it is one of the most common anti-patterns that new developers mistakenly use.

I will conclude with two more anti-patterns. The examples in this section focus on issues that are both common and nonobvious. Readers who expect to write complex or large-scale SPARQL queries are encouraged to consult additional resources and, where possible, experiment with query planners and warnings provided by their chosen tools.

SPARQL Anti-pattern 2: OPTIONAL Can Multiply Results in Unexpected Ways

A common source of confusion in SPARQL is the behavior of `OPTIONAL`. `OPTIONAL` patterns do not behave like conditional statements in programming languages. Instead, they extend existing solution mappings when a match is found. When an `OPTIONAL` pattern matches multiple values, it can multiply the number of results. This can occur even when the underlying RDF graph contains no duplicate triples.

Consider the query shown in Listing 8-12 over the People Ontology.

```
# Looks reasonable, but may produce duplicate-looking results
PREFIX :  <https://michaeldebellis.com/people/>
SELECT ?birthPlace
WHERE {:Albert_Einstein :born_in ?birthPlace.
OPTIONAL {:Albert_Einstein :has_employer ?organization.}}
```

Listing 8-12 The OPTIONAL Anti-pattern

If Albert Einstein is affiliated with multiple organizations, this query will return the same `?birthPlace` value once for each organization. From the perspective of the RDF graph, there is still only one `:born_in` triple. The duplication arises because the `OPTIONAL` pattern produces multiple solution mappings that all project to the same visible result.

In cases like this, the duplication is often accidental rather than meaningful. Adding `DISTINCT` produces the intended result (Listing 8-13).

```
# Use DISTINCT to guard against OPTIONAL anti-pattern
SELECT DISTINCT ?birthPlace
WHERE {:Albert_Einstein :born_in ?birthPlace.
  OPTIONAL { :Albert_Einstein :has_employer ?organization. }}
```

Listing 8-13 Using DISTINCT to Guard Against the OPTIONAL Anti-pattern

This example illustrates why `OPTIONAL` should be used with care, especially in exploratory queries. When `OPTIONAL` patterns can match multiple values, they can easily multiply results in ways that are not immediately obvious.

SPARQL Anti-pattern 3: Broad, Unconstrained Queries Scale Poorly

Another common anti-pattern is to write overly broad queries. The temptation to do so often arises because they may be good enough to do the job in small cases, making it easy to forget the consequences that can arise as the system scales up. For example, consider the query shown in Listing 8-14.

```
# Anti-pattern: Extremely Broad Query
SELECT *
WHERE {?person rdf:type :Person.
  ?person ?p ?o.}
```

Listing 8-14 An Extremely Broad Query

This query retrieves all triples associated with all persons in the graph, regardless of whether those triples are relevant to the task at hand. On small datasets, this may appear to work fine. On larger graphs,

however, it can produce very large result sets and place unnecessary load on the query engine.

In practice, queries like this are best treated as temporary exploration tools. Once the relevant structure is understood, they should be refined by

- Selecting only the properties of interest
- Limiting the number of variables returned
- Focusing on a specific entity rather than an entire class

For example, Listing 8-15 shows a query far more targeted.

```
# Constraining a broad query to one individual
SELECT ?p ?o
WHERE {:Albert_Einstein ?p ?o.}
```

Listing 8-15 Constraining a Broad Query to One Entity

If you don't want to constrain to one individual, use the LIMIT keyword. That way if your query returns 100,000 solutions, you will only see a representative sample (Listing 8-16).

```
# Constraining a broad query using LIMIT
SELECT *
WHERE {?person rdf:type :Person.
  ?person ?p ?o.} LIMIT 100
```

Listing 8-16 Constraining a Broad Query Using LIMIT

Heuristics for Writing Good SPARQL Queries

From these examples, we can extract a few practical heuristics that are useful when designing SPARQL queries:

- Use `SELECT *` sparingly and primarily for exploration and debugging.
- Add `DISTINCT` when broad queries may produce duplicate rows. However, remember `DISTINCT` only hides duplicate results; it does not improve performance because those duplicates are still matched.
- Prefer `CONSTRUCT` when the goal is to retrieve a subgraph rather than a table.
- Be cautious of queries with multiple triple patterns that do not share variables.
- When combining independent patterns, use `UNION` rather than placing them in the same group.
- Remember that performance issues often arise from query structure, not dataset size.
- Always add the LIMIT keyword at the end if you aren't sure how much data your query may return. All SPARQL engines have variables that limit the time a query can take, and the size of the solution set it can return. However, you don't want to hit those limits. If you do, you may see nothing and you may also add lots of temporary objects to the graph that will impede performance unless you save and reload your graph. Adding LIMIT 100 will still give you example data (so you then can write more constrained queries) but will guarantee that you don't exceed the limits of your SPARQL engine.

These guidelines help ensure that SPARQL queries remain both expressive and efficient, and they reinforce the broader design principle that queries should reflect conceptual intent as clearly as possible.

ETL, ELT, and Semantic Design

The preceding section dealt with efficient queries. We now return to the topic of data pipelines. Historically, analytic systems populated data warehouses using an Extract → Transform → Load (ETL) process, where data was cleaned and normalized before being loaded into the analytic store. Applied to Semantic Knowledge Graphs, this approach corresponds to transforming or discarding invalid values prior to ingestion so that OWL datatype constraints are never violated.

With the rise of data lakes, an alternative process became common: Extract → Load → Transform (ELT). One of the core ideas behind data lakes was to store data as it exists, deferring transformation until later

stages when the intended analysis is better understood. This approach aligns naturally with schema-on-read systems, where structure and constraints are applied when data is accessed rather than when it is stored.

For Semantic Knowledge Graphs, ELT-style ingestion offers three important advantages:

1. *Transformations can be complex and error prone*. Requiring transformation before loading means the complexity and risk of error of each transformation is now the combined complexity and risk of error for the loading process.
2. *Transformations that require data from several other objects can often be easier when the latest batch of data is already uploaded*, rather than doing it piecemeal on each data record.
3. *Loading the original raw data into the graph provides an audit trail* for the more complex objects created by transformations. For example, a standard way to deal with values such as "N/A" in a numeric field is to just delete it. However, there may be a difference in the source data between no value and "N/A". Storing the raw data enables us to trace back and understand how objects and relations in the graph were defined.

Deferring transformation also allows different consumers to impose different interpretations over the same raw data. For example, transforming a string with a list of names into instances of the Person class (and not creating multiple instances for the same person) can be an error-prone process. To do that before loading the data into the knowledge graph can make the ingestion process longer and error prone.

However, storing raw data without semantics also has well-known risks. A common mantra in the IT community is "data lakes became data swamps." As usual, there is no one right answer. In general, my experience has been the ELT approach is more appropriate for Semantic Knowledge Graphs; however, there are many use cases when ETL is best.

Categories of Data Transformations

Data pipelines apply a variety of transformations to reshape, refine, and integrate data as it moves from raw sources into a structured, semantically meaningful knowledge graph. Although vendors sometimes use inconsistent terminology, most transformations fall into well-defined categories. These help clarify what each step accomplishes and why it belongs in the pipeline. Understanding these categories is especially important in the context of Semantic Knowledge Graphs. The transformations required to convert real-world data into OWL instances and properties are often driven as much by the structure of the source data as by the target ontology.

Although the majority of enterprise data still arrives in relational form such as tables, CSV files, SQL extracts, spreadsheets, or data lake tables, modern pipelines must accommodate a broader range of formats. Operational systems frequently export nested JSON, HTML, or XML; event-driven architectures stream records via event buses such as Kafka. In addition, older systems continue to emit hierarchical or semistructured files. Regardless of the source modality, the objective of a Semantic Knowledge Graph pipeline is the same: to transform heterogeneous input into a clean, connected graph that conforms to the OWL ontology and its associated SHACL design model. For clarity, this section focuses primarily on transformations from relational representations into graph structures. These are both the most common in practice and the easiest to illustrate systematically. However, the same transformation principles apply equally to other semistructured and unstructured sources. It all comes down to reshaping data, aligning schemas, enriching with reference information, and assigning semantics.

Overview of Basic Relational Database Terminology

Because most real-world data is relational, it is useful to understand a few core relational concepts. Not at the depth required of a database specialist, but enough to appreciate how graph ingestion is influenced by

relational structure. Some readers will find this very familiar, but for some, it will be a memory refresh or new information.

The first core relational concept is the *join*, the mechanism relational databases use to reconstruct relationships that are stored indirectly. In a normalized schema, each table represents one type of database entity, and relationships between entities are represented not by explicit links but by matching key values. A join combines rows from two tables by matching these keys. These inferred connections become edges in the knowledge graph.

An *inner join* retains only rows where matching keys exist in both tables, while an *outer join* also preserves rows with missing matches, typically introducing null values. The distinction between inner and outer joins is relevant primarily because unmatched rows often signal missing or inconsistent data. Thus, once the data is represented as a graph, they become a target to tackle with SHACL shapes.

In a relational schema, each table typically has a primary key that uniquely identifies its rows. Relationships between entities are represented indirectly using foreign keys: columns whose values reference the primary key of another table. For example, an Order row may include a customer identifier that refers to the primary key of the corresponding Customer row. Foreign keys are usually modeled in OWL as object properties.

For example, a foreign key such as `customer_id` in an `Orders` table indicates that each order refers to a particular customer. In a Semantic Knowledge Graph, that column becomes an object property, such as `ordered_by`, connecting the instances of `Order` to instances of `Customer`. Much of the practical work of ingestion involves distinguishing which fields represent attributes of an entity (which will be modeled as data properties) and which encode relationships to other entities (which will be modeled as object properties). Primary and foreign keys provide strong clues.

Table 8-2 Long/Normalized Table

Pivoting is a data transformation that reorganizes data by turning values in one column into column headers, aggregating multiple rows into a more compact representation. The table that uses pairs of attributes and values is known as the *Long* or *Normalized* form. The table that results after pivoting is known as *Wide* form. Table 8-2 is an example of a long-form table and Table 8-3 is the wide-form table after pivoting. The long form is typically preferred for Online Transaction Processing (OLTP) systems, while the wide form is more useful for analytic systems.

Table 8-3 Wide Form Table After Pivot

Wide tables distribute values across many columns such as quarterly sales or sensor readings. Long tables represent each entity/attribute pair as its own row. Wide formats often obscure the regular structure of data, whereas long formats expose it and tend to map directly into RDF triples. *Unpivoting* is therefore frequently an early transformation step, since it converts data into a shape that more accurately reflects the ontology's conceptual structure and thus facilitates ingestion. Conversely, pivoting may be necessary when preparing data for downstream analytic or reporting systems that expect a particular table layout.

Relational databases rely heavily on *normalization*. Normalization refers to a family of increasingly strict design principles known as normal forms, conventionally numbered from First Normal Form (1NF) through Fifth Normal Form (5NF). As schemas move toward higher normal forms, data is decomposed into a larger number of smaller tables in order to reduce redundancy and isolate facts that vary independently. Instead of repeating information such as listing a patient's medications, conditions, and physicians all in one row, normalized schemas separate these into distinct tables linked by key values. This reduces redundancy and avoids inconsistencies, but it also means that the relationships among entities are stored implicitly and must be reconstructed through joins. In a knowledge graph, those same relationships become explicit object properties connecting instances of classes. In effect, graph modeling takes the decomposed structure created by normalization and makes the connections first-class elements of the data, eliminating the need for joins and often making the underlying conceptual structure clearer and more flexible.

With these relational concepts in mind—joins, foreign keys, data shape, and normalization—we can now turn to the transformation categories themselves. The categories that follow comprise the common operations used to convert relational, semistructured, and other practical data sources into well-formed knowledge graph structures that align with the ontology and satisfy the constraints of the design model.

Row-Level (Record-Level) Transformations

Record-level transformations operate independently on each row. No information from other rows is needed. These are the simplest and most scalable transformations because each record can be processed in isolation. Common transformations of this type include

- *Filtering*: Keep or remove rows based on logical conditions.
- *Projection*: Select, rename, or drop columns.
- *Type Conversions*: Cast strings to xsd:integer, xsd:decimal, convert informal dates to xsd:dateTime, etc.
- *Normalization and Cleaning*: Trim whitespace, standardize text, parse dates.
- *Simple Derivations*: Field-level transformations that compute new values from existing fields, such as calculating someone's age using their birthday and the current date.

Grouping and Set-Based Transformations

Set-based transformations involve multiple rows at once. These operations define relationships within a dataset rather than across datasets, including

- *Aggregation*: Compute sums, averages, counts, percentiles, and other metrics.
- *Grouping*: Partition data by keys and compute statistics within each group.
- *Sorting*: Order rows based on numeric, alphabetic, or datetime fields.
- *Window Functions*: Window functions compute values over a moving set of rows (the "window") defined by an ordering such as time, alphabetical order, or numeric ranking. Examples include running totals, moving averages, ranking, lag/lead.
- *Deduplication*: Identify and keep only the "best" record per key.
- *Sampling*: Select a subset of records for analysis or validation.

Relational (Multi-dataset) Transformations

These transformations sit at the heart of most data integration problems:

- *Coordination*: Transformations that integrate or reconcile multiple data sources.
- *Combination*: Joins to merge datasets based on key values.
- *Union/Append*: Combine datasets with the same schema.
- *Lookups/Enrichment*: Add attributes from reference or dimension tables.
- *Entity Resolution*: Match different records representing the same entity.
- *Schema Alignment*: Rename or restructure fields to fit a canonical model.
- *Conformance*: Map source codes or labels to standardized controlled lexical vocabularies.

Named Entity Recognition (NER)

Entity resolution is one of the most critical transformations in a knowledge graph pipeline because it is the step where "strings become things" (Singhal, 2012). Many source systems do not provide stable identifiers, but rather represent entities only as text: names, labels, codes, or free-form strings. During ingestion, these textual references must be matched to the correct instances in the graph or, when necessary, used to create new ones. This includes parsing compound fields (such as author lists), normalizing names, disambiguating ambiguous terms, and linking textual mentions to ontology classes or individuals. Reliable entity resolution is essential because, without it, a knowledge graph quickly fragments into duplicate or inconsistent nodes. That would undermine retrieval, reasoning, and any downstream analytics or RAG workflows. In practice, entity resolution transformations often involve a combination of string processing, approximate matching, controlled vocabularies, and contextual cues drawn from the ontology.

Taking advantage of Natural Language Processing tools is essential for successful entity resolution.[2]

Named Entity Recognition (NER) in general (not just for authors or individuals) can also play an important role in entity resolution, especially when ingesting free-text documents or semistructured content. NER models identify spans of text that refer to people, organizations, locations, and other entity types, providing an initial layer of structure that helps convert unstructured strings into candidate instances in the knowledge graph. For example, a sentence mentioning “Chomsky” and “MIT” can be automatically annotated as referring to a Person and an Organization, even before resolving the specific individuals or institutions in the ontology. Tools such as spaCy (explosion.ai, 2025) provide fast, accurate, and freely available NER models that perform well across many domains and can be extended with custom training data when needed. While NER alone cannot fully resolve identity, it offers a powerful starting point for identifying which strings in a text correspond to instances of classes that should be linked into the graph.

The knowledge graph vendor Graphwise (previously known as Ontotext) provides a tool called Tagger that illustrates the power of combining Named Entity Recognition with knowledge graphs (Graphwise, 2026). Tagger can be used in many capacities, both at run time and as a tool to help with the design and enrichment of an ontology. Tagger uses NER to provide an alternative way to browse and analyze text by identifying mentions of entities that correspond to classes and instances already present (or needing to be added) to the graph. Figures 8-3 and 8-4 show some examples of Tagger. This capability is also powerful at run time when combined with large language model architectures such as retrieval-augmented generation (RAG), which we will discuss in the next chapter.

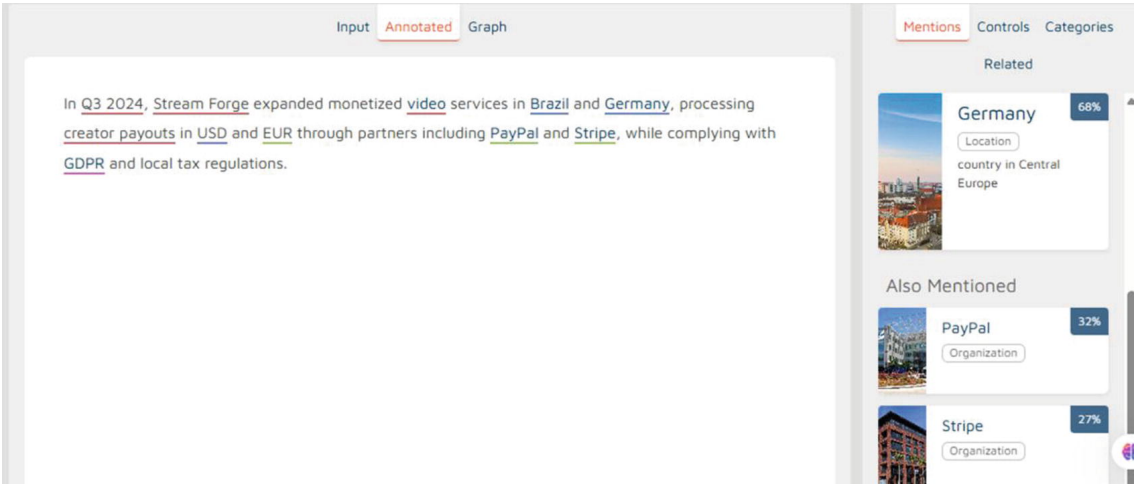


Figure 8-3 Graphwise Tagger tool for Named Entity Recognition: annotation view

In the ‘Analyzing Requirements Documents’ section of **Chapter** 6, we discussed how requirements text can be analyzed using grammatical heuristics. For example, nouns often correspond to classes or instances, and noun phrases often suggest relationships or attributes. Tools such as Tagger provide a complementary, tool-assisted approach to this analysis by automatically identifying candidate entities in natural language text and classifying them into categories such as organizations, locations, currencies, dates, and regulatory frameworks.

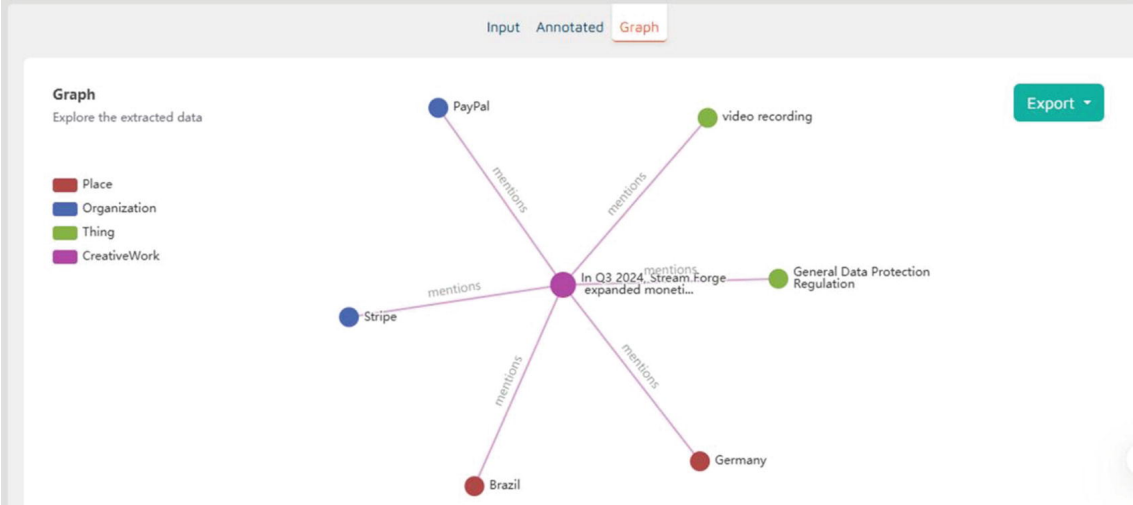


Figure 8-4 Graphwise Tagger tool for Named Entity Recognition: graph view

Figures 8-3 and 8-4 illustrate this idea using a short excerpt from a requirements document describing some StreamForge requirements. When processed by Tagger, the text reveals candidate instances (e.g., specific organizations, countries, or currencies) as well as candidate classes (e.g., Organization, Location, Currency, Regulation).

Data Quality and Validation Transformations

Validation transformations verify correctness, completeness, and constraint satisfaction. These can occur at several stages in a pipeline depending on the architecture. SHACL is useful for all the categories described here, but it is especially powerful for defining quality and validity metrics. SPARQL is one of the most powerful tools for both finding and refactoring data that does not match a quality or validity metric. These include

- *Range checks* (numeric or temporal constraints)
- *Pattern validation* (emails, phone numbers, codes)
- *Referential integrity checks*
- *Datatype checking*
- *Constraint enforcement* using SHACL, SQL constraints, or Great Expectations
- *Error tagging* or quarantining records for later review

Semantic and Business-Logic Transformations

Transformations that introduce meaning or align raw data with conceptual models:

- *Mapping* to canonical data models
- *Applying domain logic* (e.g., risk scores, category assignments)
- *Ontology-driven enrichment* (mapping values to SKOS/OWL terms)
- *Deriving classifications* or types based on patterns or rules
- *Adding semantic metadata* or lineage information (e.g., PROV-O annotations)

These transformations move from syntactic preparation to semantic understanding and are central to Semantic Knowledge Graph architectures.

Ontology Refactoring, Reuse, Validation, and Testing with ROBOT

ROBOT (OBO Foundry, 2026) is a command-line tool developed within the OBO Foundry community to support large-scale, collaborative ontology development. It was created to address practical needs shared by many ontology projects: how to automate repetitive design tasks, enforce modeling conventions, and

detect unintended changes as ontologies evolve. ROBOT supports many different types of Semantic Knowledge Graph reuse, design validation, and testing. Hence, we'll go into it in a bit of detail as a good way to cover the various types of design validation that can apply to Semantic Knowledge Graphs regardless of the tools used to validate them.

ROBOT is very useful on any ontology. However, it was originally developed to support ontology engineering practices within the *OBO Foundry*. As a result, some of its built-in validation rules reflect assumptions and conventions common in OBO-aligned ontologies.

When ROBOT is applied to ontologies outside the OBO ecosystem, it may report warnings or errors that reference internal OBO identifiers or patterns, such as IRIs of the form `OBO_Class_####`. In contexts where an ontology is not intended to conform to OBO Foundry practices, such warnings can safely be ignored. In this chapter, we focus exclusively on ROBOT's general-purpose validation and testing capabilities which are applicable to Semantic Knowledge Graph design regardless of domain.

ROBOT is implemented as a shell-based program that can be executed from the command line or integrated into automated workflows. Rather than being an interactive modeling environment, it is designed to be used as part of a *repeatable* build or validation process.

In the lifecycle described in this book, ROBOT fits within the design phase. At this stage, the conceptual model has already been established: key classes, relationships, and high-level semantics are in place. The design task is to refine that initial conceptual model into a more fleshed-out model that is

- Internally consistent
- Explicit about constraints and expectations
- Ready to be used safely by downstream systems

ROBOT supports this phase by making aspects of the design executable. Instead of relying solely on visual inspection or informal review, designers can encode their assumptions *directly as tests* and run them automatically whenever the model changes.

Importantly, these checks operate on the model itself, not on application code or runtime behavior.

At a high level, ROBOT supports four broad categories of functionality:

1. Automated refactoring
2. Ontology reuse
3. Validation and quality checks
4. Automated testing

The following sections drill down into each of these categories in turn, in each case starting with a bullet list of more specific functionalities and followed by a detailed description of each.

Automated Refactoring

Automated refactoring includes.

- Normalizing ontology structure
- Managing imports
- Converting between serializations
- Applying standardized modeling patterns

Normalizing ontology structure. This refers to applying a consistent and predictable organization to an ontology, ensuring that similar modeling decisions are represented consistently throughout the model. The problem this addresses is that, in practice, ontologies often evolve organically, with different contributors introducing slightly different patterns for naming, imports, annotations, or class hierarchies. Over time, these inconsistencies can make the ontology tremendously harder to understand, maintain, and extend. Tools such as ROBOT help fight this by automating parts of the normalization process which enforce agreed-

upon structural conventions. Examples include ensuring that imports are declared consistently, that annotations follow uniform patterns, and that redundant or legacy axioms are rewritten into a standardized form. Normalization does not change the meaning of the ontology; instead, it makes the structure more regular and easier to reason about. This benefits both humans reviewing the design and tools that process the model.

Managing imports. This controls how an ontology depends on other ontologies and vocabularies, ensuring that those dependencies are explicit, consistent, and intentional. In Semantic Web systems, imports are a powerful mechanism: they allow an ontology to reuse classes, properties, and axioms defined elsewhere without copying them. The problem addressed here is that unmanaged imports can quickly become a source of complexity and unexpected behavior.

As ontologies evolve, imports may be added informally, duplicated across files, or left in place after they are no longer needed. This can lead to problems such as unintended inferences, version mismatches, circular dependencies, or unnecessary performance overhead during reasoning and validation. Design-time tools like ROBOT can help detect and manage these situations by making imports visible, repeatable, and auditable.

From a design perspective, disciplined import management ensures that the ontology's dependencies are clear and stable. This is especially important in collaborative environments or long-lived projects, where changes in external ontologies can otherwise have hidden downstream effects.

Converting between serializations. This refers to transforming an ontology between different RDF-based syntactic formats, such as Turtle, RDF/XML, JSON-LD, or N-Triples, without changing the underlying semantics. These serializations are simply alternative ways of representing the same RDF graph, necessary because different tools, workflows, and audiences often prefer different formats.

This can easily be done with Protégé by simply loading a file in one format (e.g., RDF/XML) and then saving in another (e.g., Turtle). The limitation is scale. Protégé is an interactive ontology editor, not a graph database. Depending on memory allocation, the number of triples, and the complexity of the OWL axioms, graphs of only a few million triples (which is small by industry standards) can become impractical to load. I have seen this happen at five million triples.

Applying standardized modeling patterns. Checks that the ontology consistently uses well-understood structural solutions to recurring ontology design problems (for examples, see the ontology patterns in **Chapter** 7). Many modeling challenges appear repeatedly across domains. These challenges include representing roles, part–whole relationships, agents, organizations, and n-ary relations. Ontology design patterns capture proven ways of addressing these challenges in a way that is semantically precise and widely understood.

In practice, ontologies developed over time often end up containing multiple ad hoc solutions to the same modeling problem—especially when different contributors are involved. This not only introduces unnecessary complexity but also makes it much harder to extend or align the model with other ontologies. Tools such as ROBOT can help enforce or apply agreed-upon patterns automatically, ensuring that similar concepts are modeled in similar ways throughout the design.

From a design perspective, the value of standardized patterns is not merely stylistic. Consistent patterns improve readability, support reuse, and reduce the risk of subtle semantic inconsistencies. They also make it easier for both humans and tools to interpret the model correctly as it evolves.

Ontology Reuse

Reusing ontologies comes with very different issues than reusing code. When reusing code, we can import a library and specify which specific functions we need. Similarly, if we have code developed for a specific system, it is fairly straightforward to modify that code to make it a reusable library by removing detail from the specific initial system and (usually) making them parameters that can be passed to the reusable functions or methods.

Reusing ontologies is more complex because the intertwining between conceptual entities complicates establishing the boundaries on what is brought over. For example, suppose you only wanted to reuse the Agent pattern from PROV-O. You can't just take the `Agent` class. You need subclasses such as `Person` and

`Organization`, object properties such as `has_member` and `has_address`, and data properties such as `email`. For the Agent pattern, the specific additional classes and properties are clear. However, in more complex domains (such as the various models in the OBO Foundry that model healthcare, biology, and other related domains), understanding which entities to extract can become highly complex.

This is a very common issue with the OBO Foundry vocabularies because they cover so many intertwined and overlapping domains. That is why ROBOT has excellent tools for finding, extracting, and merging entities from one ontology into another.

Validation and Quality Checks

Validation and quality checks include

- Detection of modeling anti-patterns
- Structural checks
- Metadata completeness

Detection of modeling anti-patterns was described in **Chapter 5**.

Structural checks. Structural checks verify that an ontology conforms to basic modeling expectations and internal consistency rules, independent of any specific domain semantics. These checks focus on the shape of the model rather than its meaning, ensuring that core elements such as classes, properties, and ontologies themselves are defined in a coherent and usable way.

Examples of structural issues include missing declarations, malformed axioms, inconsistent use of annotation properties, or incomplete ontology metadata. While such issues may not cause logical inconsistencies or immediate reasoning failures, they often indicate design problems that can lead to confusion, poor tool support, or integration difficulties later in the lifecycle.

Metadata completeness. Metadata completeness ensures that an ontology includes the descriptive information required for it to be understood, reused, and governed effectively. This includes metadata about both individual entities such as classes and properties and the ontology as a whole. Common examples include human-readable labels, textual definitions, authorship information, version identifiers, and high-level descriptions of scope and intent. Protégé has preferences for automatically adding metadata every time an entity is created or edited.

Design-time validation tools can automatically detect missing or incomplete metadata and flag it early. By treating metadata completeness as a testable design requirement, ontology designers can prevent usability and governance issues from accumulating as the model evolves.

One of the most useful commands in ROBOT for ontology validation is the `report` command. When I was developing the People Ontology, I ran the `report` command to see the issues that would be found. The specific command in this example is

```
ROBOT report   --input people_ontology.ttl  --output robot_report.tsv
```

ROBOT emits reports in TSV (Tab-Separated Values) rather than CSV (Comma-Separated Values) because TSV is more robust for ontology diagnostics, avoids delimiter collisions with natural language text, and integrates cleanly with automated testing and CI pipelines. The report generated by ROBOT can be seen in Table 8-4.

Table 8-4 Results of Running ROBOT on an Early Version of the People Ontology

Automated Testing: Overview of the Most Common Technologies

- SPARQL queries as executable design assertions
- SHACL shapes as formalized constraints
- Reasoners to detect logical inconsistencies or unintended inferences

SPARQL queries as executable design assertions. Using SPARQL queries as executable design assertions means expressing expectations about an ontology in a form that can be checked automatically.

In this role, a SPARQL query functions much like an *assertion* in software design: it states a condition that should be true, and failure to satisfy that condition signals a problem in the model. Meyer's Design by Contract approach popularized this use of assertions as explicit correctness conditions in software design (Meyer, 1988).

This approach takes previously implicit design assumptions and makes them explicit and testable. It allows those assumptions to be rechecked automatically whenever the ontology changes. As a result, SPARQL-defined assertions provide a practical mechanism for regression testing semantic design models as they evolve over time. This can represent a huge savings in the effort of doing quality software engineering.

SHACL shapes as formalized constraints. Using SHACL shapes as formalized constraints means expressing machine-executable design rules about the structure and allowed content of a knowledge graph. SHACL provides a way to specify which properties are expected, which values are permitted, and how different parts of the model relate to one another, without changing the underlying OWL semantics of the ontology.

In the design phase, SHACL shapes serve as a bridge between abstract conceptual modeling and concrete implementation expectations. They allow designers to state constraints such as required properties, cardinality limits, or permitted value types in a precise and testable way. Unlike informal documentation, these constraints can be validated automatically against the model or against example data.

By treating SHACL shapes as part of the design artifact rather than as an after-the-fact validation step, designers can ensure that structural and integrity assumptions are explicit and consistently enforced. This reduces ambiguity, supports collaboration, and provides a clear contract that downstream applications and data pipelines can rely on. This offers a huge savings in software engineering overhead.

Reasoners to detect logical inconsistencies or unintended inferences. Using reasoners to detect logical inconsistencies or unintended inferences means applying automated logical reasoning to verify that the axioms in an ontology behave as intended when taken together. While individual class or property definitions may appear correct in isolation, their interaction can sometimes produce contradictions or inferences that were not anticipated by the designer.

Logical inconsistencies arise when the axioms of the ontology cannot all be true at the same time. For example, when a class is defined in a way that makes it unsatisfiable. Unintended inferences, on the other hand, occur when the logical consequences of the model go beyond what was intended, such as inferring class memberships or relationships that do not align with the domain meaning.

In the design phase, reasoners are used not to validate data, but to validate the structure and semantics of the model itself. Running a reasoner as part of an automated design check allows designers to catch these issues early, before they are embedded in downstream applications or relied upon by users. This helps ensure that the ontology's formal semantics accurately reflect the intended conceptualization of the domain.

Custom SPARQL Testing

In addition to automated tools like ROBOT, it is possible to simply write Linux, Windows, or Python scripts that use SPARQL to automate validation and testing. Although ROBOT can also run custom SPARQL queries, writing your own shell scripts gives you many more options regarding how to control the ordering and output of the queries. In the directory https://github.com/mdebellis/SemanticKG-Design/tree/main/tests/verify

I've created a collection of SPARQL queries for common validation checks and a Linux shell script called `run-verify.sh` that runs them. More information can be found in the README file in that directory.

Exposing the Design Model to Distributed Systems

The design model does not exist in isolation. Semantic Knowledge Graphs must integrate with a wide range of distributed systems, including applications, services, data pipelines, and analytics platforms.

This integration is achieved through well-established interface patterns such as REST APIs, GraphQL services, remote procedure calls (RPC), and event or message-based systems. These are not semantic technologies. They are the mechanisms through which the semantic model is exposed, accessed, and operationalized within an enterprise architecture.

From a design perspective, the key point is: *the Semantic Knowledge Graph defines the meaning of the data; the integration pattern defines how that data is accessed and exchanged.*

The design model therefore plays a dual role. On one hand, it translates conceptual semantics into executable structures such as SHACL shapes and validation logic. On the other, it implicitly defines the contracts that external systems will use to interact with the data product. Table 8-5 provides an overview of the most common technologies used to design interfaces among distributed systems.

Table 8-5 Standards for Integrating Distributed Systems

Design Implications

Each of these patterns exposes the same underlying semantic model in a different way. The choice of pattern should be driven by the needs of the consuming systems rather than by the internal representation of the data.

For example:

- A REST API may expose a Person resource with endpoints such as /persons/{id}, reflecting a resource-oriented view of the data.
- A GraphQL interface may allow a client to request a Person along with related entities (e.g., employer, publications, or relationships) in a single query, reflecting the connected nature of the graph.
- An event stream may publish updates when new instances of a class are created or when key properties change, enabling downstream systems to react in near real time.

In all cases, the design model ensures that these interfaces are grounded in a consistent semantic foundation. Class definitions, property constraints, and validation rules defined in SHACL shapes can be used to guide API schemas, validate incoming data, and enforce governance policies across integration points.

These integration mechanisms do not replace the semantic model; they depend on it. As we'll see in **Chapters 11 and 13**, Semantic Knowledge Graphs can define the semantics and dependencies for enterprise data. This includes the way that data is accessed between systems that don't share the same memory space.

A Note on GraphQL and Semantic Models

GraphQL is often seen as a natural fit for graph-based data because it allows clients to traverse relationships and retrieve nested structures in a single query. This can make it particularly effective for exposing data derived from a knowledge graph.

However, GraphQL schemas are not equivalent to ontologies. They define the structure of queries and responses, but they do not capture the full logical semantics expressed in OWL, such as class hierarchies, disjointness constraints, or inference rules.

For this reason, GraphQL should be viewed as a complementary layer:

- The ontology defines what is true about the domain.
- The design model defines how that truth is validated and structured.
- GraphQL (or REST, or events) defines how that information is accessed.

The design model serves as the bridge between semantic representation and system integration. By grounding APIs, service interfaces, and event streams in a shared semantic foundation, organizations can ensure that distributed systems remain consistent, interoperable, and aligned with the domain model.

Internationalization As a Design Constraint

Internationalization is a good example of how design modeling differs from conceptual modeling. Internationalization does not change meaning; it changes how that meaning is exposed to users, tools, and applications across linguistic and cultural contexts. For that reason, internationalization is an excellent example of a design requirement.

Internationalization can be extremely important because it directly affects governance, stewardship, and long-term control of semantic assets. As ontologies grow beyond their original context, they are often reused by multiple teams, exposed through catalogs, or integrated into enterprise governance frameworks. In these settings, labels and descriptions are not merely convenience features; they become part of the governed surface of the knowledge graph. Without explicit language tagging, it becomes difficult to manage accountability, ensure consistent interpretation, or meet organizational and regulatory expectations for clarity and accessibility.

From a governance perspective, language-tagged labels support traceability and policy enforcement. They make it possible to define and validate expectations such as "all public classes must have labels in at least one approved language" or "controlled vocabularies must provide definitions suitable for nontechnical stakeholders." By encoding these expectations directly in the design model, internationalization becomes something that can be tested, audited, and enforced over time.

In the European Union and elsewhere, linguistic diversity is a legal and policy requirement, particularly for public-sector information and cross-border services. While EU regulations do not mandate a specific technical representation, they create strong governance expectations that metadata and semantic descriptions be accessible across languages, making internationalization a practical design requirement for many semantic systems.

Language Tags

Human-facing annotations (e.g., `rdfs:label`) should carry language tags. These are short strings like `@en` (for English) or `@fr` (for French). Getting into the habit early pays off later when you need to conform to internationalization standards such as i18n (Phillips et al., 2025).

One gotcha that initially surprises people: a language-tagged literal (e.g., `"Albert"@en`) is ***not*** an `xsd:string`. In RDF 1.1, untyped "simple literals" are treated as if they were `xsd:string`, but language-tagged literals are a distinct datatype called an `rdf:langString`. There is a drop-down menu in Protégé you can use to define a tag for any string. When you do that, the string automatically becomes an instance of the `rdf:langString` datatype. Note that this means in SPARQL:

```
" Albert"@en ≠ " Albert"^^xsd:string
```

It's better to normalize labels early. Luckily, you can do this automatically with SPARQL. The update in Listing 8-17 will convert any untagged `xsd:string` labels into language-tagged ones. The nice thing is that these updates don't even need your ontology IRI, so they're reusable across projects. All you need to do is to change the `"en"` to whatever language tag you need and the following queries should convert labels from string to `rdf:langString` and add the language tag you specify. This update uses a SPARQL operator called `VALUES`. SPARQL's `VALUES` clause temporarily binds a variable to a fixed list of permitted values. It works much like a Python for loop that iterates over a small, explicit list. For example:

```
for p in ["rdfs:label", "skos:prefLabel", "skos:altLabel"]:
```

except that SPARQL applies the variable bindings declaratively when matching graph patterns. Note: as with many SPARQL features, you can't do this with the SPARQL engines that currently work with Protégé; they don't support `VALUES`. The update in Listing 8-17 below will convert any untagged `xsd:string` labels into language tagged oneslanguage-tagged ones.

```
# Example: apply normalization to add lang tags to rdfs:label,
# skos:prefLabel, and skos:altLabel
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
DELETE {?s ?p ?old }
INSERT { ?s ?p ?new }
WHERE { VALUES ?p { rdfs:label skos:prefLabel skos:altLabel }
  ?s ?p ?old.
  FILTER( lang(?old) = "" || datatype(?old) = xsd:string )
  BIND(STRLANG(STR(?old), "en") AS ?new)}
```

Listing 8-17 SPARQL Updates to Change Labels from strings to langStrings

The filters deliberately exclude literals that already have a language tag, so you don't overwrite multilingual content. This makes the query idempotent, which means you can run it repeatedly and it will update any new data since the last time you ran it but won't change anything it shouldn't.

OWL Profiles and Reasoning Trade-Offs

OWL is often described as a single language, but in practice, it is better understood as a family of profiles that impose different constraints on how the language can be used. These profiles exist to balance expressive power, computational complexity, and tool support. Choosing an OWL profile is therefore a design decision, not a conceptual one, and it has direct consequences for performance, reasoning guarantees, and interoperability.

At a high level, OWL profiles fall into two categories:

1. Standard profiles defined by the W3C
2. Product-specific profiles defined by individual vendors

Both are important to understand.

W3C-Defined OWL Profiles

The W3C defines several official OWL profiles, each designed to support a particular set of features that are described below.

OWL DL

OWL DL (Description Logic) is the profile most commonly assumed when people say they are "using OWL." It enforces a clean separation between classes, properties, and individuals, and it restricts how constructs can be combined so that reasoning remains decidable and well-defined. Most OWL reasoners and modeling tools implicitly assume OWL DL, and it provides the strongest balance between expressiveness and reliable reasoning.

OWL EL

OWL EL is designed for very large ontologies with deep class hierarchies, such as biomedical terminologies. It sacrifices some expressive features (e.g., universal quantification and disjunction) in exchange for highly scalable reasoning. If your primary requirement is fast classification over tens or hundreds of thousands of classes, OWL EL is often a good choice.

OWL QL

OWL QL is optimized for query answering over large volumes of instance data stored in relational databases. It supports ontology-based data access (OBDA) scenarios, where SPARQL queries are rewritten into SQL. OWL QL limits expressiveness in order to guarantee efficient query execution.

OWL RL

OWL RL is designed for rule-based reasoning engines. Its constructs can be implemented using forward-chaining rules, making it suitable for systems that rely on rule engines rather than full Description Logic reasoners.

Each of these profiles represents a different point in the design space. The key takeaway is that no single profile is "best" in the abstract. The right choice depends on your nonfunctional requirements such as the scale of your data, the maximum response time, and the capabilities of your target platform.

OWL Full

OWL Full removes many of the constraints imposed by OWL DL, allowing classes to be treated as individuals, properties as classes, and so on. While this flexibility may appear attractive, it comes at a steep cost: OWL Full has no complete, sound, and terminating reasoner.

This means that once an ontology falls into OWL Full, standard OWL reasoning guarantees no longer apply. Reasoners will either refuse to reason over OWL Full ontologies, silently ignore axioms, or downgrade to incomplete reasoning without warning.

For this reason, OWL Full should be viewed not as an advanced feature, but as a design failure mode. In production systems, OWL Full is almost always entered accidentally, typically through errors such as having a data and object property with the same IRI or asserting non-annotation properties on classes without defining a pun for the class (recall a pun is an `owl:NamedIndividual` with the same IRI as a class or property).

OWL Full should be avoided in Semantic Knowledge Graph design. If meta-modeling is required, it should be done in a way that does not result in OWL Full such as using puns or annotation properties.

Product-Specific OWL Profiles

In addition to the official W3C profiles, most commercial and open-source platforms define their own practical OWL profiles. These profiles are not formal standards, but rather documented subsets or extensions of OWL that reflect what a given product can reason over efficiently.

For example, AllegroGraph supports OWL reasoning, but it requires a specific subset of OWL features that work well with its reasoning and materialization strategies. Other platforms make different trade-offs, depending on whether they emphasize forward chaining, backward chaining, query rewriting, or hybrid approaches. For example, Stardog supports the SWRL standard. In contrast, AllegroGraph provides an implementation of Prolog for rule-based reasoning rather than supporting SWRL.

This means that "OWL compliance" is not binary. Two systems may both claim OWL support while behaving very differently when presented with the same ontology. From a design perspective, this reinforces an important principle: *always design against the actual reasoning capabilities of your target platform, not against an abstract notion of OWL.*

In practice, this usually means documenting which OWL features are intentionally used, which are avoided, and why.

Profiles As a Design-Time Constraint

From a methodology standpoint, OWL profiles function as guardrails. They constrain what can be expressed, but in doing so, they make systems more predictable, scalable, and maintainable.

In this book, the assumption is that most production Semantic Knowledge Graphs will

- Stay within OWL DL or a well-understood subset
- Avoid OWL Full entirely
- Be designed with explicit awareness of platform-specific reasoning behavior

These choices allow OWL-based systems to function reliably in large-scale, integrated environments.

A common misconception is that an ontology explicitly declares which OWL profile it uses (e.g., OWL DL or OWL RL). In fact, the OWL specification defines profiles syntactically: an ontology belongs to a profile

only if it satisfies that profile's constraints. There is no formal mechanism for declaring a profile within the ontology itself.

In practice, tools such as Protégé and DL reasoners (e.g., Pellet or HermiT) help enforce these constraints by reporting errors when an ontology violates OWL DL restrictions. These errors are often prominently displayed, making it clear that something is wrong. However, not all environments provide such safeguards. Systems that use rule-based reasoning may accept ontologies that fall outside OWL DL without warning, potentially leading to subtle inconsistencies or loss of expected reasoning behavior. This is another reason to create SPARQL queries or SHACL constraints that test boundary conditions on axioms. They can help you catch scenarios where your reasoner doesn't support a specific feature that you use.

In the next chapter, we will see how these design constraints interact with SHACL, validation, and testing, and how OWL and SHACL together form a practical foundation for enterprise-grade semantic systems.

Summary

- *Design modeling marks the transition from meaning to robustness*, shifting the focus from idealized conceptual correctness to how Semantic Knowledge Graphs behave under real-world conditions such as incomplete, inconsistent, and evolving data.
- *Semantic Knowledge Graphs are designed to coexist with heterogeneous data architectures*, complementing relational databases, analytic platforms, and streaming systems rather than replacing them.
- *Basic graph-theoretic concepts such as trees, DAGs, and networks are essential design concepts*. Because graph-theoretic concepts are utilized pervasively by Semantic Web languages and tools, it is worth the effort to understand the definitions of the most common types of graphs.
- *Named graphs are an essential design tool.* They can define design boundaries such as encapsulation and security.
- *SPARQL graph operations make data movement and promotion explicit*, enabling clean ingestion pipelines that move data from raw to validated to published states.
- *Data ingestion is a design activity*, because it is where ontology assumptions first collide with real data and where failure modes become visible.
- *The StreamForge ingestion example demonstrates how data, provenance, and process context coexist*, showing how ingestion runs, domain data, and metadata form a unified, queryable model.
- *SHACL can be a derived design artifact, expressing validation intent* already present in the ontology rather than serving as a parallel or competing model.
- *SPARQL enables systematic OWL-to-SHACL transformations*, allowing datatype expectations and functional-property assumptions to be operationalized without duplicating semantic intent.
- *ETL vs. ELT choices have semantic consequences, and deferring transformation (ELT) is often better aligned with Semantic Knowledge Graph design* because it preserves provenance, flexibility, and auditability.
- *Data transformations in Semantic Knowledge Graph pipelines fall into recognizable categories*, including row-level, relational, structural, semantic, and validation-focused transformations.
- *Named Entity Recognition and resolution are critical semantic transformations*, turning strings into shared entities and preventing graph fragmentation in real-world data.
- *Design validation and testing tools such as ROBOT, SPARQL, SHACL, and reasoners make design intent executable*, enabling regression testing, quality assurance, and controlled evolution of Semantic Knowledge Graphs.
- *Internationalization is a design concern* affecting governance, usability, and regulatory compliance rather than meaning; language-tagged annotations should always be used and should be treated as first-class, testable design requirements.
- *OWL profiles represent design-time trade-offs, balancing expressiveness, performance, and reasoning guarantees*; OWL DL or disciplined subsets should be preferred, while OWL Full should generally be avoided.

- *Further Reading. Learning SPARQL* by Bob DuCharme (DuCharme, 2013) is an excellent book to go into more detail about SPARQL.

Footnotes

1 The term *vocabulary*, similar to *entity* and *logical*, is one where different communities assign different meanings. In the ontology community, vocabulary simply means an ontology such as PROV-O designed to be a reusable asset to define technical and domain conceptual models. In the semiformal or taxonomy world, a vocabulary is a structured collection of words, their definitions, and relationships, intended to support shared understanding, communication, and information within an organization or community. We will examine the synergy between these types of models and Semantic Knowledge Graphs in **Chapter 12**. I've been using and will continue to use the term vocabulary to mean a reusable ontology. However, it will also be useful to discuss the original semi-formal meaning as well, so I will always differentiate that meaning using the term *lexical vocabulary*.

2 A real-world example: in the first retrieval-augmented generation (RAG) systems I developed, we had metadata that included an author string for documents. These typically had the format: "Church, Alonzo; Turing, Alan". However, there were an incredible number of edge cases such as "J.R.R Tolkien," "Church, Turing," "Alonzo Church, Alan Turing," and more. I handled about 80% of the author strings with a simple Python function that first did a split using the semicolon, then another split using the comma. I've never seen a better example of the last 20% requiring 80% of the work. I kept adding code for edge cases, and the result was some very ugly code. In our next RAG system, we used the Python library Nameparser and it was exponentially easier and more reliable.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_9

9. Implementation

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

OWL is the most important language when defining the conceptual model. SHACL serves that role for the design model. Implementation is when we need to select one or more programming languages.

Python is widely used for data science and Semantic Knowledge Graphs. However, in the early days of the Semantic Web, Java was the dominant language. As a result, many foundational tools, such as Protégé, were built with Java. Both are excellent languages with mature development environments.

Python has the advantages of being easier to learn and providing broader coverage through an enormous ecosystem of libraries. While Java supports modular reuse through packages and Maven repositories, Python's pip package manager makes reuse particularly effortless.

Other languages you may come across include

- *R*: A statistical computing language popular among data scientists, especially for statistical modeling and data visualization. R excels at exploratory analysis and has an extensive ecosystem for statistics, but it's less suited for large-scale applications or general-purpose programming.
- *Rust*: Rust is a systems language that offers performance closer to C/C++ while reducing many common memory-management errors. Instead of relying on a garbage collector as Java and Python do, Rust uses ownership, borrowing, and compile-time lifetime checks to

determine when memory can be safely released. This makes it attractive for high-performance infrastructure code, although its learning curve is steeper than that of Java or Python. Rust is valued for reliability and speed, particularly in infrastructure software and performance-critical systems, including components used in AI/ML systems.

- *JavaScript/TypeScript*: JavaScript is the standard language for user interfaces running in browsers. Many graph visualization libraries and front-end interfaces for knowledge graphs (e.g., D3.js, Cytoscape.js) are written in JavaScript or its typed variant, TypeScript.
- *Scala*: A language that runs on the Java Virtual Machine (JVM) and combines functional and object-oriented paradigms. Scala can use Java libraries such as Apache Jena and is used in Big Data platforms such as Apache Spark.
- *Lisp*: Although it has largely fallen out of the mainstream, Lisp was the first, and for many years, the dominant programming language for AI and other advanced systems. You may still encounter it occasionally, particularly in specialized or legacy systems. A modern descendant of Lisp worth mentioning is Clojure, which runs on the Java Virtual Machine and combines Lisp's functional expressiveness with seamless access to the Java ecosystem.

Python and Java both offer mature libraries for accessing Semantic Knowledge Graphs. This book focuses primarily on Python.

As discussed earlier, there are two very different approaches to accessing a knowledge graph from a programming language:

1. *Use a graph database client*. This gets you persistent data, transactions, security, and additional useful tools as well as *clients* in Python, Java, and other languages. A graph database client is a software environment that is a client of the database in the sense of a client/server architecture.
2. *Use a library*. There are libraries in Python, Java, and other languages that allow you to connect your knowledge graph to a

programming language and use that language to load data, create a user interface, etc.

Both approaches have their place, and on large projects, it is possible to use both. Next, we discuss the criteria for choosing between a triplestore graph database client (e.g., the AllegroGraph Python client) and an embedded library (e.g., RDFLib for Python).

Triplestore Graph Database Client

Recall that a triplestore is a dedicated database server that manages RDF data (a.k.a. triples), exposes a SPARQL endpoint, and usually provides reasoning, indexing, transactions, and enterprise-grade features. Major triplestore products ship with client software development kits (SDKs) for Python, Java, and other languages.

Vendors may offer free or community editions for evaluating their triplestores. The included client libraries, RDF/SPARQL and reasoning capabilities, graph visualization tools, and resource limits vary by product and edition. These editions can be useful for learning and prototyping, but their feature sets and limits should be checked against the intended workload.

License terms for community editions vary by vendor and may include *significant licensing restrictions*. Check the current terms to determine whether an edition *permits production use*. Also check capacity, user, time, and hosting limits; an operational deployment may require a commercial license or a different edition.

Apache Jena is an open-source library with many essential tools for working with Semantic Knowledge Graphs. It is also commonly used, particularly when an open-source solution is required. Jena provides an RDF and SPARQL stack with persistent storage and server components. Its suitability for a deployment depends on the workload, configuration, and operational requirements.

Strengths of using triplestore clients are

- Scales to enterprise datasets (billions of triples).

- Supports multiuser concurrent access.
- Provides reasoning, security, backups, and monitoring.
- Some triplestores support vector embeddings and LLM integration alongside graph data; the available features depend on the product and edition.
- Free or community editions can provide a useful environment for an initial Semantic Knowledge Graph prototype. Check the vendor's current license, capacity limits, and any time or inactivity restrictions.
- Fits naturally into enterprise patterns such as Data Lakehouse, Data Mesh, or Data Fabric.

Using a triplestore is often the best choice for enterprise data platforms that need reliability, strong security, and scale. Our illustrative example in this book is primarily AllegroGraph from Franz Incorporated. This is not an endorsement of AllegroGraph. It was easiest to focus on one tool, and AllegroGraph is the tool I have the most experience with at this time. When using AllegroGraph with Python, their Python client provides excellent tools to interact with the knowledge graph and to use the myriad other libraries and tools in Python.

Embedded Libraries

An alternative to a triplestore client is to work entirely in-process with a library. Popular choices include RDFLib and Owlready2 for Python and Apache Jena (which has a library that can be used independently of the Jena graph database) and the OWL API for Java.

Strengths of embedded libraries:

- Zero server dependency; no need to install and configure infrastructure
- Free and open source
- Easy to distribute with tools, CLIs, or plugins

- Great for generating documentation, validating ontologies, running tests, or experimenting

This is best for projects that want a small footprint. Examples include projects creating developer tools, build-time utilities, single-user apps, or any situation when asking end-users to install a triplestore would be too onerous. For example, most of the coding examples on the GitHub site that supports this book use RDFLib for that reason.

Leading Embedded Libraries

Python:

- ***RDFLib*** is a widely used Python library for working with RDF data and SPARQL. It provides in-memory and file-based stores, making it ideal for lightweight applications or development environments that don't require a full graph database. The API supports parsing, serializing, querying, and manipulating triples, along with utilities for namespaces, graphs, and datasets. Although it supports basic RDFS reasoning through plugins, it does not implement a full OWL reasoner or SHACL engine.
- ***Owlready2*** is designed specifically for manipulating OWL ontologies in Python, including both TBox and ABox content. It supports automatic synchronization between Python objects and OWL individuals, which makes it convenient for integrating ontologies with Python-based applications. The library includes a built-in link to the HermiT reasoner (via Java), providing OWL 2 DL reasoning capabilities. While powerful for ontology-centric workflows, it is not optimized as a general RDF graph engine and is less suitable for large datasets. Owlready2 is ideal when Python developers need tight, object-oriented integration with OWL ontologies. This is useful when you want to store data as OWL individuals while defining methods on the corresponding Python classes.

Java:

- ***Apache Jena*** is a widely used Java framework for RDF, RDFS, OWL, and SPARQL, offering a comprehensive open-source programming environment and multiple storage backends. It includes support for SPARQL 1.1, TDB/TDB2 persistent stores, a rule-based reasoner, and a variety of tools for parsing, inference, and RDF processing. Jena is also the foundation for several production RDF systems and scales well for enterprise applications. While it includes reasoning capabilities, the built-in reasoner is rule-based and not a full OWL 2 DL reasoner. Jena is typically used when organizations want a flexible, scalable Semantic Web stack within Java.
- ***The OWLAPI*** is a Java library focused explicitly on editing, analyzing, and reasoning over OWL ontologies rather than general RDF graph management. It serves as the foundation for major OWL tools such as Protégé and supports all OWL 2 profiles, serialization formats, and structural transformations. Because it integrates with leading OWL reasoners (ELK, HermiT, Pellet), applications can use it to access OWL reasoning through those reasoners. It does not provide a SPARQL engine or RDF store, so it is usually paired with another system when querying large datasets. The OWLAPI is ideal for ontology-engineering tasks, transformations, analysis, and tooling.

Cloud Landing Zones

In modern IT environments, it is very common to utilize a cloud environment such as Amazon Web Services or Microsoft Azure for development and deployment. Before any Semantic Knowledge Graph, data pipeline, or application component can be deployed in a hosted environment, organizations typically establish what is known as a *landing zone*. A landing zone is a preconfigured cloud foundation that provides the baseline infrastructure, security, and governance required to deploy and operate systems safely and consistently at scale.

Landing zones are not products that are purchased and installed. Instead, they are architectural environments that are designed and configured, often using vendor-provided reference architectures,

templates, or automation frameworks. The concept originated in large public cloud platforms, but it is best understood as a general architectural pattern rather than a vendor-specific feature.

A typical landing zone includes

- Account or subscription structures that separate environments such as development, testing, and production
- Identity and access management (IAM) policies defining who can access which resources
- Network configuration, including virtual networks, subnets, and connectivity rules
- Centralized logging, monitoring, and auditing
- Baseline security controls and policy enforcement

In most organizations, the responsibility for defining and maintaining the landing zone falls to platform, infrastructure, or cloud governance teams rather than individual application or data teams. Application teams typically deploy into an existing landing zone that supplies foundational controls for security, networking, and governance. They remain responsible for the application's configuration, data-access controls, and operating requirements.

The landing zone defines the operational context in which the system will run. Hosted graph databases, vector indexes, ingestion pipelines, and language-model integrations all assume the existence of secure storage, controlled network access, and reliable monitoring, i.e., capabilities provided by the landing zone.

Hosted offerings are also available; check each vendor's current deployment options, supported features, and terms.

Introducing landing zones at this stage of the lifecycle reinforces an important architectural principle: *implementation concerns extend beyond code and data schemas*. They include the deployment environment and the operational controls it provides. By standardizing these foundational elements, landing zones reduce friction between teams, improve security posture, and support incremental system

evolution. This aligns very well with the Agile, model-driven approach described throughout this book.

Examples of Common Ways to Write Code for Semantic Web Systems

This section will walk through two illustrative examples of common types of code written for Semantic Knowledge Graphs: parameterized SPARQL functions and retrieval-augmented generation (RAG) systems.

Parameterized SPARQL Functions

Parameterized SPARQL functions are among the most common types of functions developed for Semantic Knowledge Graphs. If you've worked with relational databases, you've probably developed similar functions for SQL. SPARQL is a very powerful query language but too technical for most users. Parameterized SPARQL functions provide a way to make your queries easier, faster, and more reliable. The three most common use cases for these kinds of functions are:

1. You find yourself reusing the same SPARQL query again and again; however, every time you use it, you need to change some hard-coded strings (e.g., IRIs). This is a sign you should write a parameterized function, so you don't have to manually recreate your queries every time you use a new ontology. Instead, you pass the various strings to a function that merges them with a SPARQL query and executes the query.
2. You have users that you don't expect to know SPARQL, but you still want them to be able to query the knowledge graph. In this use case, you create a Graphical User Interface (GUI) with menus, text fields, etc. The user sets the GUI widgets to find what she wants, and those parameters are passed to a function that generates and runs a SPARQL function with the appropriate user parameters.
3. You have several SPARQL queries that you run on a regular basis, e.g., as part of regression testing, and you want to run them all at

e.g., as part of regression testing, and you want to run them all at once rather than manually one at a time.

The core idea behind a parameterized SPARQL function is that the function has some hard-coded text strings for parts of the query that always remain the same and variables bound at run time for the parts of the query that change. The function takes in parameters and puts them in the appropriate place in the query. That is, the function generates one or more queries at run time and executes them.

To see a simple example, think back to **Chapter 3**, where we used SPARQL queries to generate labels for ontologies. The SPARQL queries for the People Ontology are in the file: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/SPARQL_Label_Generation_Updates_People.ru

For example, the first CONSTRUCT query to generate labels for classes is shown in Listing 9-1.

```
CONSTRUCT {?class rdfs:label?lblname.}
WHERE {?class rdf:type owl:Class.
    BIND(STRAFTER(STR(?class), 'people/') as ?
name)
    BIND(REPLACE(?name,"_", " " ) as ?lblname)
    OPTIONAL{?class rdfs:label ?elbl.}
FILTER(?class != owl:Thing && ?class !=
owl:Nothing && "" != ?lblname && !BOUND(?elbl))}
```

Listing 9-1 SPARQL `CONSTRUCT` Query to Generate Labels for Classes in the People Ontology

Recall that I created a small example ontology based on some of the analysis patterns in Fowler's book on object-oriented analysis patterns (Fowler, 1997), and in that ontology, I didn't define any labels. To use the queries from the People Ontology for the Pattern Ontology, we had to create a new file with the same SPARQL queries and change the string `'people/'` to `'pattern_examples/'` in each query. We had

to modify five queries: for classes (shown in Listing 9-1), object, datatype, and annotation properties and for individuals. The result is the file https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_3/Listing_3-14_SPARQL_Label_Generation_Updates_Patterns.ru.

Then, we had to run each `CONSTRUCT` query in Snap SPARQL. These queries return triples; incorporating the returned triples into the ontology is a separate step in that workflow. It is not a ton of work, but it is still worth asking: Can we improve the process? With a parameterized SPARQL function, we can.

This section shows how to automate that process using a simple Python script. The goal is not to replace SPARQL with Python logic, but rather to demonstrate a common and useful implementation pattern: keep SPARQL queries as explicit text but generate and execute them programmatically by parameterizing the parts that change.

This is the same pattern used in many systems, including applications where SPARQL queries are generated from GUI forms.

The script https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_9/add_labels.py performs the following steps:

1. Takes an ontology stored as a Turtle file (.ttl).
2. Generates SPARQL Update (`INSERT ... WHERE`) statements to add missing `rdfs:label` annotations for classes, object properties, datatype properties, annotation properties, and named individuals. Here, RDFLib uses `INSERT` statements to modify the graph directly. In contrast, `CONSTRUCT` queries return triples; incorporating those triples is a separate step in the Snap SPARQL workflow.
3. Executes those updates using RDFLib.
4. Writes a new ontology file whose name is the original filename that has “with_labels” appended.

Parameterizing the SPARQL UPDATE

The file `add_labels.py` provides a simple example of wrapping parameterized SPARQL updates in a function that supplies the appropriate values at run time.

In earlier examples, the namespace of the ontology is hard-coded directly into each SPARQL update. In particular, the updates assume a specific namespace when extracting the local name of a class or property from its IRI.

For example, the following expression assumes that all entities we want to add labels to use the `...people/` namespace segment:

```
BIND(STRAFTER(STR(?class), 'people/') AS ?name)
```

This works as long as the ontology namespace remains the same. However, when the same label-generation logic is applied to a different ontology, the user must change the namespace string (e.g., replacing `'people/'` with another ontology-specific segment). Everything else —the triple patterns, filters, and label-generation logic—remains identical.

If we rely solely on SPARQL, this means manually editing the same string in multiple queries. That is why we had two different files in **Chapter 3** with almost identical SPARQL queries. Making these changes by hand is tedious and error-prone and makes reuse more difficult.

The approach used in `add_labels.py` avoids this problem by treating SPARQL updates as string templates and supplying the parts of the update that must be changed, such as the ontology namespace, as parameters to the function. This makes the queries reusable across ontologies while keeping the SPARQL logic itself unchanged.

This accomplishes three things:

1. We don't have to write new SPARQL updates for each new ontology or IRI. Instead, we just pass the IRI as a parameter to the function.
2. Rather than manually executing up to five different updates (classes, object properties, data properties, annotation properties

(classes, object properties, data properties, annotation properties, and named individuals), we run one function that executes every update.

3. The function can be added to larger scripts that do testing and automatic refactoring of knowledge graphs.

In addition, we can further increase reusability by adding additional optional parameters for other parts of the query. For example, we can add a language-tag parameter and support any legal language, not simply English.

SPARQL Templates As Strings

In Python, each SPARQL update template (e.g., `CLASS_LABEL_INSERT_TEMPLATE` in `add_labels.py`) is simply a string variable. It contains the full `SPARQL UPDATE` text, written exactly as it would appear in a SPARQL update, except that certain values are written as placeholders, such as `{namespace_segment}`, `{delimiter}, and {lang}`.

These templates are not executable SPARQL yet. They are generic patterns that describe how labels should be generated, independent of any particular ontology. The SPARQL logic remains fully visible and readable.

Packaging Parameters with a Dataclass

Rather than passing several independent parameters around (`namespace segment`, `delimiter`, `language tag`), the example groups them into a single object called `LabelQueryParams`.

This class is defined as a dataclass, using Python's `@dataclass` decorator. A dataclass is an ordinary Python class; the decorator generates methods such as `__init__ and __repr__`, and the class can also define its own methods. Here, it groups the configuration values used to specialize the SPARQL templates for a particular ontology.

Rendering Templates into SPARQL

The function `render_update()` takes two inputs: a SPARQL template string and a `LabelQueryParams` instance. Its only job is to substitute the parameter values into the template, producing a complete `SPARQL UPDATE` string with no placeholders remaining.

At this point, the output is exactly the same SPARQL that would have been written manually. Nothing new is added to SPARQL itself; the Python code simply automates the substitution step that would otherwise be done by hand.

Generating and Organizing Updates

The function `generate_label_updates(input_graph, namespace, delimiter="_")` applies this rendering step to each of the SPARQL templates. The results are stored in a Python dictionary (a key-value structure similar to a hash map or associative array), where the keys are human-readable names and the values are the rendered SPARQL updates.

This organization serves two purposes. First, it makes the execution logic straightforward: the program can simply iterate over the dictionary and apply each update in turn. Second, it makes debugging and inspection easier: the same dictionary can be written out to a file so that the generated SPARQL can be reviewed independently of the Python code.

This pattern is well suited to graphical user interfaces and other tools where users are able to benefit from SPARQL-based queries without being required to write or edit SPARQL themselves.

In such systems, the SPARQL templates capture the semantics of the operation, while the parameters capture the user's intent. With appropriate parameter validation and escaping, the application can generate valid SPARQL while keeping the query structure visible for inspection.

Idempotence and Safety

Each SPARQL update is written to be idempotent, meaning that it can be run multiple times without changing the result after the first successful execution. For example, if an entity already has an `rdfs:label`, no new label is added. This is how I use these queries on an almost daily basis in my own work. I'll create an initial ontology with user-defined IRIs in Protégé. When you use that option, Protégé leaves the label undefined. When I have enough entities created, I'll run the SPARQL updates and generate labels. Then, as I continue to work and add more entities, I can rerun the updates to generate labels for them without creating duplicate labels.

Another important consideration is that there isn't always a one-to-one mapping between the user-generated IRI and the label. For example, sometimes the label will include a copyright or trademark character. In those cases, labels need to be written manually. The function must also preserve existing labels. We don't want to overwrite manually created labels.

Running, Debugging, and Inspection of Generated SPARQL

In practice, it is often useful to inspect the exact SPARQL being executed, either for debugging or simply to understand what the program is doing. For this reason, the script includes an optional --debug flag. Without debugging, the script only outputs the new ontology file. With debugging enabled, the script also writes all generated SPARQL updates to an `.ru` file alongside the output ontology. This reflects a common best practice: even when queries are generated and executed programmatically, it should always be easy to view or copy the actual SPARQL text.

To run the function, first clone the GitHub repository for the book to your local machine. Then, navigate to the directory containing `add_labels.py` and run the following command in the shell:

```
python add_labels.py
Pattern_Examples_No_Labels.ttl --ns
"pattern_examples/" --debug
```

The function will generate a new ontology file in the same directory the function is in (make sure that the input file is in that directory as well) and will append the string "with_labels" to the original file name. The result in this example will be a file called `Pattern_Examples_No_Labels_with_labels.ttl` as well as a file called `Pattern_Examples_No_Labels_queries.ru` that shows the specific queries that the function generated. The `ns` parameter is the string to use that indicates the IRIs you want to add labels to. The `delimiter` is the character to use for blanks (in this case "_"), and `lang` is the code for the lang strings.

Why This Pattern Matters

This pattern is extremely general:

- The SPARQL queries themselves are explicit and readable.
- The variability is isolated into a small set of parameters.
- The same logic can be reused across ontologies, environments, and tools.
- Multiple updates can be run as one batch.

This is the same approach used in larger systems where SPARQL queries are assembled dynamically based on configuration files, APIs, or user interface input. The label generator shown here is intentionally modest, but it illustrates a technique that scales directly to more complex query generation tasks.

For example, a common application of this pattern is to develop user interfaces that execute SPARQL but don't require the user to understand the SPARQL language. In DeBellis et al. (2023), we provided users with a GUI developed with the Qt Python library to search the knowledge graph for Non-Governmental Organizations (NGOs) in India. They could use the GUI to select specific states in India, specific NGO services, the size of the NGO, and other relevant information. That information was then passed from Qt into a SPARQL template to create a complete SPARQL query. That query was run in the client of our graph

database, and the results were presented back to the user in a scrollable list.

Retrieval-Augmented Generation (RAG)

The previous example illustrated how Semantic Knowledge Graphs integrate with traditional programming workflows. We now turn to a contemporary integration pattern: retrieval-augmented generation (RAG). A RAG architecture supplements a large language model (LLM) with retrieved evidence. This can improve the grounding and inspectability of answers, but it does not eliminate hallucinations. The model still relies on learned weights, and retrieval similarity is not a measure of factual correctness. Researchers at Google DeepMind made partial progress in explaining a specific factual-recall task in Pythia-2.8B while leaving some mechanisms unresolved (Nanda et al., 2023). This illustrates a challenge in understanding how particular answers are generated, rather than a universal impossibility of locating factual information. Citations retrieved or generated for an answer can help users assess its support, but they do not identify the training examples that caused the model to produce it.

A RAG system addresses this problem by supplementing the LLM's knowledge base with a document corpus for a specific domain. As with all architectural decisions, this benefit comes with a trade-off. Although an LLM can respond across many topics, a RAG system's retrieved evidence is limited to its selected corpus. A system may be configured to answer only within that domain. Lewis et al. (2020) combined a pretrained sequence-to-sequence model with a dense vector index of Wikipedia accessed through a pretrained neural retriever. Some graph databases also support vector retrieval, allowing graph data and a document corpus to be used together in a RAG system.

Of course, a vector as an abstract data structure is merely an ordered list of numbers that can be stored in a one-dimensional array. The challenge lies not in storing vectors but in using them efficiently. Embedding vectors represent text as points in a model-dependent vector space. Efficient retrieval over a large corpus often uses indexes

and nearest-neighbor search rather than comparing the query against every stored vector. This retrieval workload differs from the matrix computations used to run the embedding and generation models. Graph databases that offer vector retrieval combine these capabilities with graph storage and querying. The databases that support knowledge graphs as well as vector databases are known as *neurosymbolic databases* (Sheth et al., 2023) to emphasize support for vectors (neuro) and knowledge graphs (symbolic) (Clark, 2024; Popov, 2025; Hunger et al., 2021).

The code examples for this section are more complex than for the parameterized SPARQL functions above. Also, it is more difficult to find specific examples that can stand alone from the complete system. As a result, I'm going to provide URLs to GitHub repositories for previous RAG systems I've built for examples, but I won't go through the detailed instructions for running the code in this section as I've done in previous sections. The two previous RAG systems that I'll discuss were in the domains of Dental Materials (DeBellis et al., 2024) and Climate Obstruction (DeBellis et al., 2025). The GitHub repository for Dental Materials is at https://github.com/mdebellis/DrMO_Docs and for Climate Obstruction at https://github.com/mdebellis/Climate_Obstruction.

In addition, I created an example Data Catalog for use in **Chapters 11 and 13** that can be found here: https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/knowledge_graph/streamforge_data_catalog_V1.ttl. I also developed a RAG to show how RAG can be used to search and utilize documents for an Enterprise Data Catalog. I'll use that as an example in the following sections and will discuss it in more detail in **Chapters 11 and 13**.

Graph RAG Data Pipeline

Figure 9-1 shows the data pipeline for ingesting documents into a neurosymbolic knowledge base using a RAG architecture. Each major task in the pipeline has a number reflecting when it happens in the

process. Each arrow represents data flow. Figure 9-1 explains the details of ingesting documents into a RAG architecture. The next section shows how to use those documents at run time with an LLM.

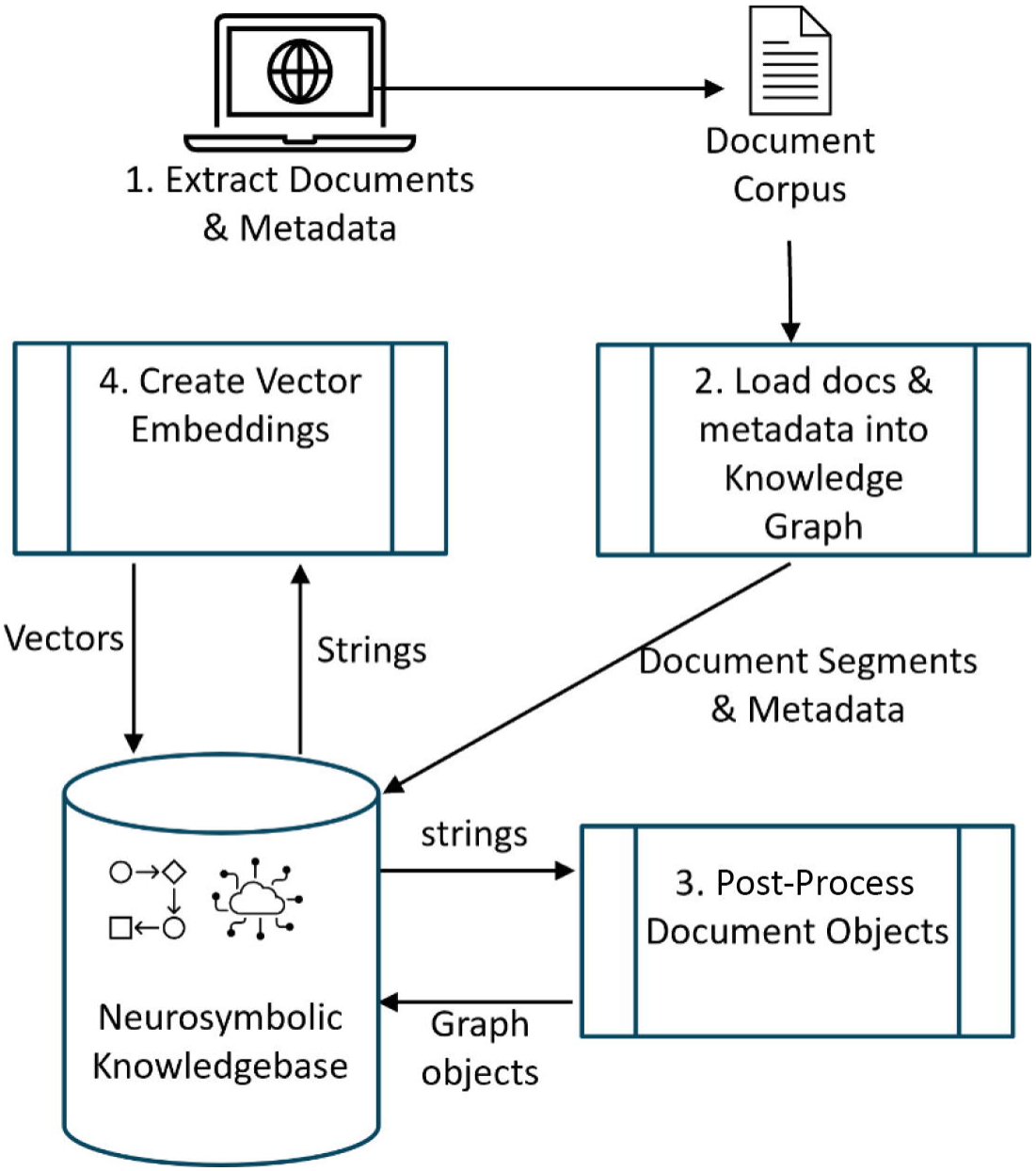


Figure 9-1 Data pipeline for retrieval-augmented generation corpus

Task 1: Extract Documents and Metadata

In this task, we define the types of documents that we want in our corpus. This requires us to define the specific Internet, Intranet, or

other sources, the metadata used to determine which documents from the sources we want, and additional data describing the criteria to recheck the source for new documents. Examples of criteria include

- Checking sites on a time interval (daily, weekly, monthly) for new or edited pages that match the search parameters
- Looking in a directory for new files that have been deposited by other extraction systems
- In event architectures, defining consumers that match parameters of events created by producers

For example, in DeBellis et al. (2024), the sources were sites that had medical journal articles such as PubMed. The additional metadata consisted of a set of search queries defined by the project's domain expert. These queries were designed to identify documents related to dental materials, which was the focus of the RAG system in that example. In another example (DeBellis et al., 2025), the sources were websites used by social scientists performing research in Climate Obstruction. In this project, we used the Prefect Python data pipeline library. In this phase, document metadata is stored as a row in a CSV file.

Task 2: Load Documents and Metadata

In this step, we create the initial knowledge graph objects to store the documents and metadata. This task consists of two subtasks:

1. *Create a document instance*. We create an instance of the `Document` class or one of its subclasses such as `Web_Page` or `Journal_Article`. This object will be the main entry point for all the metadata as well as the text of each document.
2. *Divide documents into chunks*. LLMs represent text meaning as large vectors called embeddings. Before embeddings are generated, documents in the corpus are divided into smaller segments, a process known as chunking. Rather than creating a single vector for an entire document (which dilutes meaning and exceeds practical model limits), each document is broken into chunks of moderate

size, typically on the order of 100 to 500 words. Those values are best understood as heuristics rather than fixed rules: the optimal chunk size depends on the embedding model, the nature of the source material, and the kinds of queries the system is expected to support. Smaller chunks tend to improve retrieval precision by keeping vectors tightly focused on a single idea, while larger chunks preserve more context but risk blending unrelated topics.

Beautiful Soup (Richardson, 2007) can parse HTML documents so that application code can extract text and divide it into chunks. The chunking rules determine which passages are used to create embeddings. The file that did this for the Climate Obstruction RAG is https://github.com/mdebellis/Climate_Obstruction/blob/main/src/parse_documents.py.

Note	In that file, we call "chunks" "segments."

Task 3: Post-Process Document Objects

This next task in the pipeline is highly domain-specific. In this task, we utilize the domain knowledge in the ontology to do further processing on each document object. This will add additional individuals and object-property assertions as well as provide other types of additional information. Examples of post-processing include

- *Turn author strings from metadata into objects and object properties*: One common type of metadata is a string listing the authors of a document such as "Turing, Alan; Church, Alonzo." In this task, we find the delimiters for each author as well as their first and last names. For each author, we search the knowledge graph for an instance of the `Person` class with an equivalent first and last name. If no such instance exists, we create one. Then we add an object property value, such as `has_author`, that points from the `Document` instance to the `Person` instance. Matching names alone can conflate different people; ambiguous matches require additional identity evidence or

review. For an example of this type of post-processing, see the file https://github.com/mdebellis/Climate_Obstruction/blob/main/src/parse_authors.py.

- *Create* `has_topic` *links*: We analyze each document using Natural Language Processing (NLP) techniques such as bag of words, stemming, and soundex (Bird et al., 2009) to create indices matching ontology entity labels with the text in each document. This creates links between each document and the entities in the ontology that it discusses. The following file uses the AllegroGraph Free Text Index (described in **Chapter 13**) to create `has_topic` links: https://github.com/mdebellis/Climate_Obstruction/blob/main/src/fti_match.py.
- *OWL reasoner*: The OWL reasoner can add additional information to the graph. For example, in the StreamForge domain, an important concept is knowing which entities are upstream or downstream of which other entities. This is modeled by two inverse superproperties: `is_upstream_of` and `is_downstream_of`. Properties such as `consumes_dataset` and `uses_dataset` both indicate a downstream relation. With the required subproperty and inverse axioms and a reasoner that supports those inferences, if `Foo consumes_dataset Bar`, then `Foo is_downstream_of Bar`.
- *Temporal reasoning*: We can utilize vocabularies such as the W3C Time vocabulary to implement Allen's interval algebra for temporal relations that apply to instances of the `Event` class such as `contains`, `overlaps`, and `precedes` (DeBellis, 2023). This requires rules, queries, or reasoning support in addition to the vocabulary itself.
- *Generate embedding notes*: This is an example of post-processing used in the RAG systems I've developed. An LLM does not automatically have access to the knowledge captured in the ontology and Semantic Knowledge Graph; that knowledge must be provided through the integration. I use a method for providing selected graph facts as text that I call *embedding notes* (DeBellis, 2026). Using simple

pattern matching as well as intuitive names that follow a standard, it is possible to generate text strings that capture much of the meaning in the graph in a way that the LLM can understand. The code for this is in the directory https://github.com/mdebellis/SemanticKG-Design/tree/main/data_product/data_product_src, as it plays a role in the Data Catalog and RAG prototype in the later chapters. Currently, there are two functions:

https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/data_product_src/add_embedding_notes_from_properties.py
https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/data_product_src/add_embedding_notes_from_classes.py

The first function `add_embedding_notes_from_properties.py` takes as input a Semantic Knowledge Graph file and a file with a list of properties in the knowledge graph that have data we want the LLM to have. For each property, the function finds all the triples with that property as a predicate. Then, for each of those triples, it generates a string that is added to the `embedding_note` annotation property. For object properties, that string will follow the pattern:

```
{Subject Label} {Predicate Label} {Object Label}
```

For datatype or annotation properties that have literals as values, it will follow the pattern

```
{Subject Label} {Predicate Label} {Predicate Value}
```

For example, in the StreamForge data catalog, the data product `Creator_Data_Product` has an `is_owned_by` value that is the

`Creator_Analytics_Team`. Since the `is_owned_by` property is in the file of properties that need embedding notes, the following string is generated and placed as an `embedding_note` on `Creator_Data_Product` (the subject of the triple): `"Creator Data Product is owned by Creator Analytics Team."` Then, when embedding vectors are created in the next task, the `embedding_note` property is one of the properties whose value generates a vector. If the user asks the RAG "who owns the Creator Data Product?" the RAG can retrieve that statement to support an answer, subject to the retrieval settings and the LLM's generation. We'll see this in action in **Chapter 13**. The `add_embedding_notes_from_classes.py` function works in a similar way. It takes a list of classes and, for each class, adds the appropriate `embedding_note` values to relevant instances and to sub-/superclasses such as "Creator Data Product is a Data Product" and "Data Product is a kind of Catalog Asset."

My original plan was to use an NLP library in Python such as spaCy. However, my colleague Dr. Robert Neches encouraged me to first try simple pattern matching, and since my knowledge graphs follow standards, e.g., properties are always all lowercase and use the "has {property name}" and (for the inverse) "is {property name} of", simple pattern matching works very well. There are still additional issues that could best be resolved with NLP libraries to further enhance this, such as digging into the RDF that defines axioms on classes and creating natural language strings for them. However, this simple pattern-matching approach works for much of the most important knowledge in the graph, making it accessible to the LLM.

Task 4: Create Vector Embeddings

In this task, we invoke an embedding model via an API to create vector embeddings for the text chunks and embedding notes prepared in **Tasks 2** and **3**. Each vector represents a piece of text in the model's vector space. At run time, we use a compatible embedding model and configuration for the user's prompt so that its vector can be compared

with the stored document vectors. The LLM that generates the final response can be a different model.

When using a graph database, there is usually some type of user interface to make this easy. Figure 9-2 shows the AllegroGraph GUI for creating vector embeddings. A common approach is to select predicates whose literal values contain text suitable for embedding. This shows creating vectors for the StreamForge Data Catalog that will be discussed in later chapters. At the top, we see that the knowledge graph being used is `streamforge_data_catalog` and then below that the vector database that holds the embeddings is `streamforge_vdb`.[1] The three IRIs at the bottom, including `skos:definition`, are IRIs in the data catalog of properties that will have strings appropriate for vector embedding. Note that, as discussed at the end of **Task 3**, the property `embedding_note` is one that will generate vectors for each string.

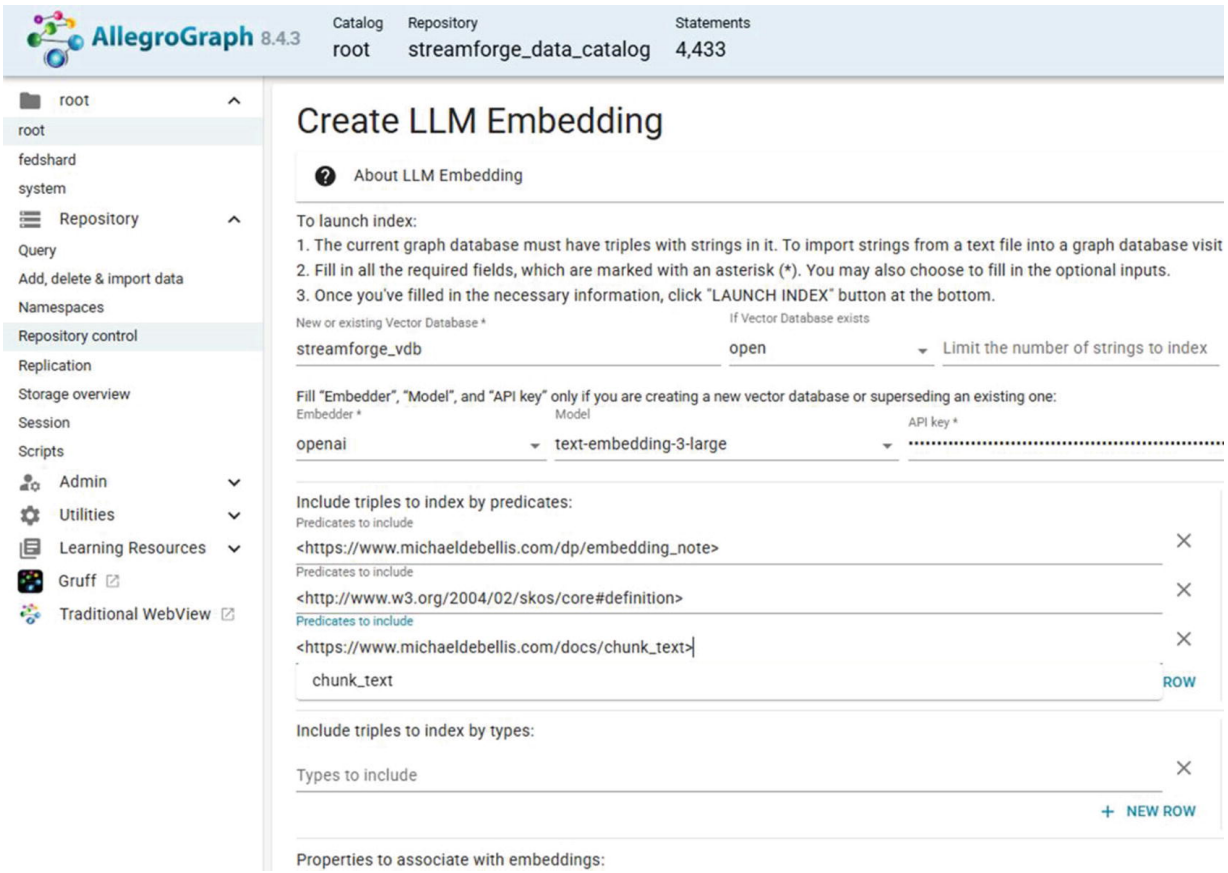


Figure 9-2 AllegroGraph GUI to generate vector embeddings

Graph RAG Runtime Architecture

This section describes the runtime data flow in a Semantic Knowledge Graph RAG system shown in Figure 9-3. The system starts with the user creating a prompt in natural language as they would with a standard LLM (**Task 1**).

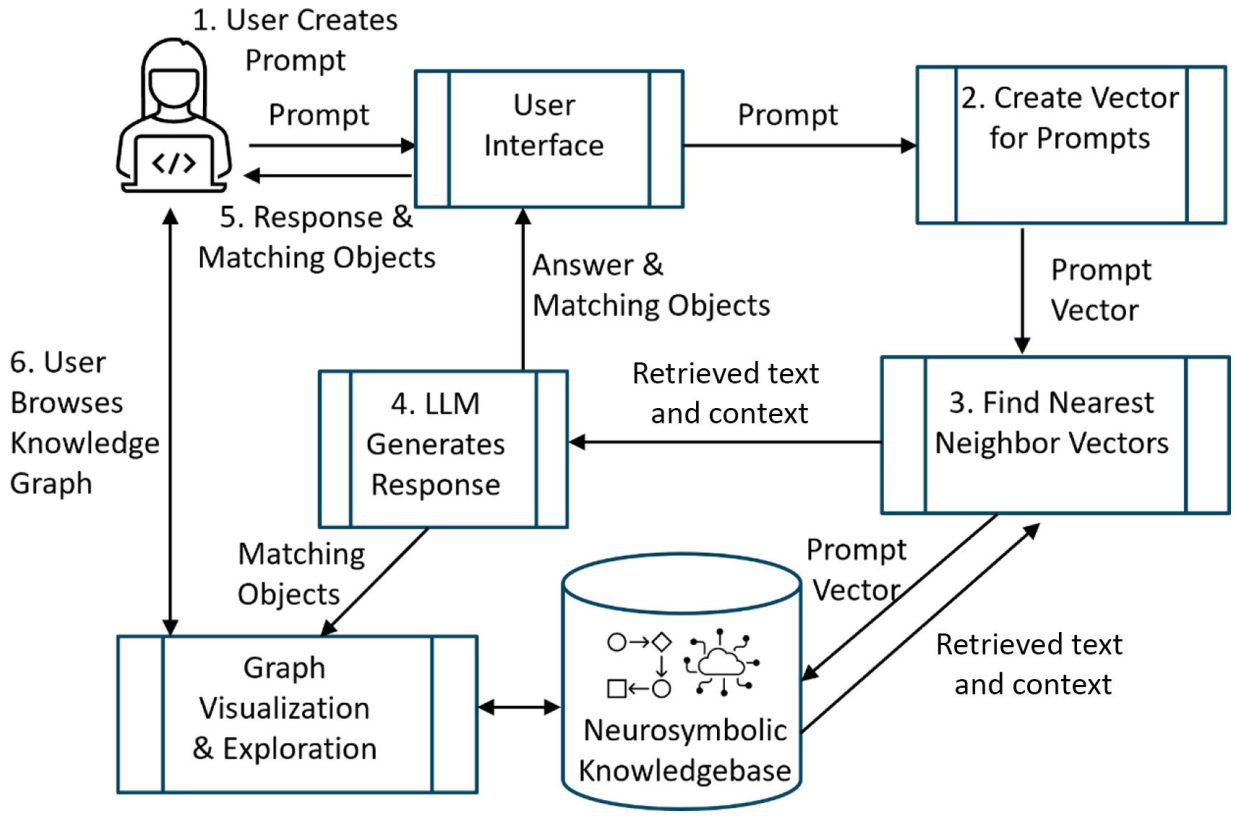


Figure 9-3 Graph RAG runtime architecture

Task 2: Create Vector for Prompt

In this task, the Semantic Knowledge Graph system calls an embedding model through an API to generate a vector for the user's prompt. The prompt embedding must use a model and configuration compatible with those used for the document embeddings in the data ingestion pipeline.

The reason for this requirement is that vector similarity is only meaningful when all vectors are created using the same embedding

model. Different models produce vectors in different vector spaces, with different dimensions and distance characteristics. If the prompt vector and the document vectors are generated using different models, similarity comparisons between them will produce unreliable or meaningless results.

Task 3: Find Nearest Neighbor Vectors

In this task, the vector for the prompt is returned to the Semantic Knowledge Graph. The system searches for corpus vectors close to the prompt vector, using the store's supported similarity measure and search method. Cosine distance is one such measure (Anastasiu & Karypis, 2016); nearest-neighbor or approximate-nearest-neighbor algorithms perform the search. The dimensions and score interpretation depend on the embedding model and database configuration. *Semantic distance* is a useful metaphor for similarity in that vector space, but it is not geographic distance or a measure of factual correctness. One of the parameters the user can set controls the *semantic distance* that is considered close enough to merit using a text chunk as relevant to the prompt. The other parameter the user sets determines how many vectors to use if more than one is within the nearness threshold (DeBellis et al., 2024).

Task 4: LLM Generates Response

The vectors themselves are not sent to the language model. Instead, the system retrieves the natural-language text associated with the matching vectors and injects that text into the LLM's prompt (along with the original user prompt). This supplies context to help ground the response in retrieved content, but the generated answer still needs to be checked.

This separation between vector-based retrieval and text-based generation is a defining characteristic of RAG systems. We can also improve the quality of our answer by having text templates that are injected into the beginning of the prompt, which can differ depending on the specific question asked or task the user is trying to solve. These text templates are strings that set the overall context such as "You are a

dental clinician who needs to find the best dental product for your patients. You need to balance cost of a product with product features and risks."

Task 5: Response and Matching Objects Returned to User

In addition to the LLM response, additional data from the knowledge graph is returned to users to provide them with additional knowledge graph objects that they can explore. This is in addition to the objects that represent each text segment and the document it appears in. Examples of relevant objects returned are the values of `has_author` and `has_topic` object properties. These and other objects can provide context for the user to further explore the knowledge graph.

In this AllegroGraph implementation, a SPARQL query using database-specific extensions works with a Python function to coordinate prompt embedding, retrieval, and response generation. The following file is part of the Climate Obstruction system. It creates a simple UI using Streamlit (Snowflake Inc., 2024). This is another example of a parameterized SPARQL function: https://github.com/mdebellis/Climate_Obstruction/blob/main/src/ui_v2.py

This creates a simple text-based user interface. When the user enters a new prompt, that is passed to the function `do_query(question)`, which finds the matching vectors, passes the strings that matched along with the user prompt to ChatGPT, and returns an answer.

This function also uses a very handy utility library called Pyperclip. Pyperclip is a Python library that can put strings on the clipboard. We use Pyperclip to save the generated SPARQL query on the clipboard. Pyperclip accesses the clipboard on the machine running the Python process. A remotely hosted app needs a separate way to transfer text to the user's browser clipboard. This is useful in two ways:

1. *Debugging*: If the query doesn't work as expected, the developer has access to the generated SPARQL.

2. *Further Browsing*: In the UI, there is a link to invoke the Gruff graphical browser from AllegroGraph. The user can invoke this, paste the generated SPARQL, and further explore the knowledge graph to gather additional information as described in **Task 6**.

Task 6: User Browses Knowledge Graph

In this final step, the user can further explore the knowledge graph. For example, in DeBellis et al. (2024), the user can see the authors for a specific paper, recognize a specific author who the user believes has done additional useful research, and expand the graph to find other papers by that author. The user can also take advantage of other tools to visualize and explore the knowledge graph. Examples include

- *Graph visualization tools* to further explore the initial graph returned with the response to the prompt. Examples of such tools: Stardog Explorer, Graphwise GraphDB Visual Graph, TopBraid EDG, AllegroGraph Gruff, and Neo4j Bloom
- *NLP tools* that provide traditional keyword searching of the knowledge graph
- *SPARQL tools and SPARQL graphical browsers* that simplify using SPARQL for nontechnical users

---

Why Graph RAG Matters

In **Part 2** of this book, we will examine a range of enterprise analytic data architecture innovations, including Data Products, Data Catalogs, Data Mesh, and Data Fabric. Despite their differences in type of analytic data, these approaches share a common requirement: *effective search*. Data is only valuable if people can discover it, understand it, and trust it. Olesen-Bagneux argues that a data catalog is a search system at its core and that the user interface of an ideal Data Catalog will resemble little more than a simple, Google-style query box (Olesen-Bagneux, 2023).

With Graph RAG, the future that Olesen-Bagneux describes has arrived. By combining knowledge graphs with retrieval-augmented

generation, organizations can apply the natural-language understanding capabilities of large language models directly to their enterprise data, while retaining access to the structure, governance controls, and provenance recorded in those systems. Graph RAG can allow users to ask questions in natural language and supply curated documents, metadata, and explicit graph relationships as context for the response. At the same time, it does not replace existing access mechanisms such as keyword search or SPARQL queries, but complements them, offering multiple modes of interaction over the same underlying knowledge assets.

This is why Graph RAG is especially compelling in the context of enterprise architectures. It enables a unified discovery experience across data products, catalogs, and distributed domains, while supporting transparency, traceability, and control when retrieval, access, and citation mechanisms are implemented and checked appropriately. In **Part 2**, we will revisit Graph RAG in a concrete implementation, showing how it can serve as a practical and low-friction entry point for bringing LLMs into enterprise data systems without turning those systems into opaque black boxes.

Summary

- *This chapter transitions from design to implementation*. The goal is to ground the reader in pragmatic decision-making by describing different programming tools and examples that illustrate some common patterns used by systems that utilize Semantic Knowledge Graphs.
- *There are two techniques for developing code that integrates with Semantic Knowledge Graphs: libraries and graph database clients*. Graph database clients excel when persistence, ACID transactions, and support for multiple concurrent users are required. Libraries are best when the power of a database is not needed and the application footprint needs to remain lean and simple.

- *Graph database clients come with all the major triplestore RDF graph database products*. These libraries allow developers to connect to the graph database, run SPARQL, and leverage libraries for data analysis in Python and other programming languages.
- *Cloud landing zones* are becoming ever more important as more and more enterprise development moves to the cloud using vendors such as Amazon Web Services and Microsoft Azure. A landing zone provides a common set of tools that all developers use as a foundation. This makes the consistency, reusability, and maintenance of development and implementation environments far easier and less costly to manage.
- *Parameterized SPARQL functions* are among the most common types of functions developed for Semantic Knowledge Graphs. These functions take SPARQL queries that can be highly reusable by changing a few lines of text such as hard-coded IRIs and make them easier to reuse by making the strings that need to be changed parameters to a function. Developing such a function alleviates the burden of rewriting the SPARQL for each different use case. It also allows us to package several SPARQL queries into one function. We can add additional parameters for debugging, language tags, etc. We used the simple example of the label-generation SPARQL queries described in **Chapter 3** to show how using this pattern makes SPARQL queries easier to reuse. A common use for this pattern is as part of a user interface to give users who don't understand query languages the power of SPARQL. Users select parameters specific to the domain from a GUI, and those parameters are passed to a SPARQL template.
- *Graph retrieval-augmented generation (RAG)* is one of the most common types of systems implemented with Semantic Knowledge Graphs. RAG supplements an LLM's pretrained knowledge with retrieved material from a selected corpus. A system can expose the passages it retrieved and use thresholds to abstain when no sufficiently relevant material is found. These mechanisms improve inspectability and can reduce unsupported answers, but similarity

scores do not establish factual correctness, and generated answers can still contain errors. A system may be configured to answer only within a selected domain, but that restriction is a design choice rather than an intrinsic limit of RAG.

- *Utilizing a knowledge graph allows more advanced context and features to be added than those provided by early RAG systems*. Examples include temporal reasoning, understanding specific domain models, and additional search tools such as “bag of words” and stemming.
- *The ability to search is a crucial requirement to manage and utilize enterprise data,* as will be discussed in more detail below. This makes RAG a powerful and appealing model for developing analytic systems such as data lakes, catalogs, and knowledge management systems.
- *Further Reading*: The W3C Semantic Web Primer for Object-Oriented Software Developers (Pan & Uschold, 2006) is an excellent overview of the similarities and differences between object-oriented design and programming and the Semantic Web model of classes, properties, and instances.

Footnotes

1 In later work, I found that it worked better for this use case to include the vectors in the knowledge graph repository rather than create a separate repository for them. Either can be used.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_10

10. Governance, Security, and Enrichment

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

This chapter concludes **Part 1** by addressing three cross-cutting concerns that apply across the entire data life cycle: governance, security, and enrichment. These concerns span across life-cycle phases including requirements, modeling, implementation, operations, and ongoing evolution. For this reason, they need to be treated together not as isolated topics, but as interdependent aspects of a coherent system design.

Governance and security are often treated as external constraints imposed after a system has been built. In practice, this approach rarely works well. When governance is deferred to the end of a project, it becomes a source of friction rather than a mechanism for enabling controlled change. If not considered from the start, friction due to intra- and inter-organizational conflicts over ownership and responsibility is further compounded by conflicts over the time, cost, inconvenience, rigidity, and inefficiency of solutions retrofitted on top of an existing software base.

In Semantic Knowledge Graph systems, governance shapes how models evolve, how data products are exposed, and how knowledge is safely enriched over time. Governance is the center of the life cycle. It defines who may create, modify, and publish semantic assets; how changes are reviewed and approved; and what policies apply to data as

it evolves. When governance is modeled explicitly, it becomes possible to support iteration without sacrificing consistency, accountability, or trust. Furthermore, it makes decisions easier to modify should those policies change in the future.

Security follows naturally from governance. Governance establishes intent, what should be accessible, by whom, and under what conditions. Security mechanisms enforce those decisions at run time. In semantic systems, security requirements can be explicitly modeled via OWL and SHACL.

Enrichment depends on governance and security. Enrichment activities introduce new knowledge into the graph via inference, validation, classification, automated augmentation, and user feedback. Data users describe the data's value, issues, aggregate metrics, etc. Without clear governance and provenance, enrichment can quickly lead to uncontrolled growth, loss of trust, and/or semantic drift. When properly governed, however, enrichment becomes a powerful and safe mechanism for increasing the value of data products over time.

Throughout this chapter, governance is treated as something supported by semantic models rather than merely enforced manually through process or documentation. Existing vocabularies play an important role in this approach. Dublin Core provides widely adopted mechanisms for describing ownership, versioning, and descriptive metadata. PROV-O enables explicit representation of provenance, lineage, and derivation. ODRL extends this foundation by providing a standard way to represent policies, including permissions, prohibitions, and duties associated with data assets and other governed resources. Together, these vocabularies allow governance information to be embedded directly into the knowledge graph and integrated with models of different domains.

The goal of this chapter is not to provide an exhaustive treatment of governance frameworks, security architectures, or enrichment techniques. Instead, it aims to show how Semantic Knowledge Graph systems support *governance by design*, enabling controlled access, traceable change, and responsible enrichment across the data life cycle.

Governance

In Semantic Knowledge Graphs, governance is a structural property of the system. In many data initiatives, governance lives primarily in documentation, review boards, or policy statements outside the technical artifacts themselves. These are all highly likely to fall by the wayside when, as often happens, deadlines, resource limitations, or unexpected complications impinge on development timelines. In contrast, with Semantic Knowledge Graphs, governance is modeled directly. This is an example of what Gartner calls active metadata (Zaidi & Thoo, 2019). This makes responsibilities, policies, and life-cycle states explicit, queryable, and enforceable. That level of automation, in turn, becomes a defense against these considerations being sacrificed to project pressures.

Governance answers a set of recurring questions: who is responsible for a data asset, who may change it, how those changes are reviewed, and how they are communicated to downstream consumers. These questions arise throughout the life cycle, from modeling and implementation to operations and enrichment. Modeling governance explicitly allows change to evolve in a controlled manner rather than through ad hoc decisions.

A key advantage of semantic systems is that governance can be represented using the same constructs as domain knowledge. Ownership, stewardship, versioning, approval status, and life-cycle state become explicit relationships rather than implicit assumptions. Once captured in the graph, governance itself can be queried: which datasets are approved for production, which models are under revision, or which artifacts are subject to regulatory constraints. **Chapter 13** demonstrates this approach through a data product catalog model built from widely adopted W3C vocabularies. I'll also use some examples from the data catalog in this chapter.

Governance is especially important in modern iterative environments that practice continuous integration and continuous delivery (CI/CD). Removing the traditional "big bang" for new releases

of a system means that the checks and balances that used to accompany each new release must be baked into the ongoing process and semantics of the knowledge graph. Explicit governance enables iterative development and CI/CD that conforms to policies and regulations. Without that explicit support, governance considerations are all too easy to neglect.

This perspective aligns with treating data as *managed assets* rather than passive datasets. Data is an evolving artifact with defined ownership, intended use, and contractual expectations. Governance provides the structure that allows such artifacts to remain reliable components of a larger system even as their internal representations change.

Explicit governance clarifies responsibilities and boundaries, making decentralized development more feasible. Teams may own different data, but shared governance models make those boundaries visible and navigable. This is an essential property in domain-driven and data product architectures which we will discuss in later chapters. It is key to building and maintaining modular systems.

Security mechanisms enforce governance decisions about access and protection, while enrichment depends on governance to determine which transformations are acceptable and how their results are evaluated and leveraged.

Semantic Policy Models for Governance (ODRL)

One of the most concrete ways governance intent manifests is through explicit policies modeled in the knowledge graph. Many governance decisions ultimately take the form of policies: statements about who may access a resource, what actions are permitted, and under what conditions those actions are allowed. In Semantic Knowledge Graph systems, policy intent can be represented explicitly using semantic models, making governance rules visible, inspectable, and queryable.

The W3C ODRL (Open Digital Rights Language) (ODRL Community Group, 2018) provides a standardized vocabulary for expressing such policies in a machine-readable form. Originally developed to describe

digital rights and usage constraints, ODRL provides a semantic policy language. It provides a small set of concepts for representing permissions and prohibitions over assets, independent of any specific enforcement mechanism.

Figure 10-1 shows the core ODRL policy model. A `Policy` acts as a container for one or more `Permissions`, `Prohibitions`, or `Duties`. Each `Permission` specifies an `Action` that may be performed on a `Target` asset by an `Assignee`. `Targets` correspond to data assets.

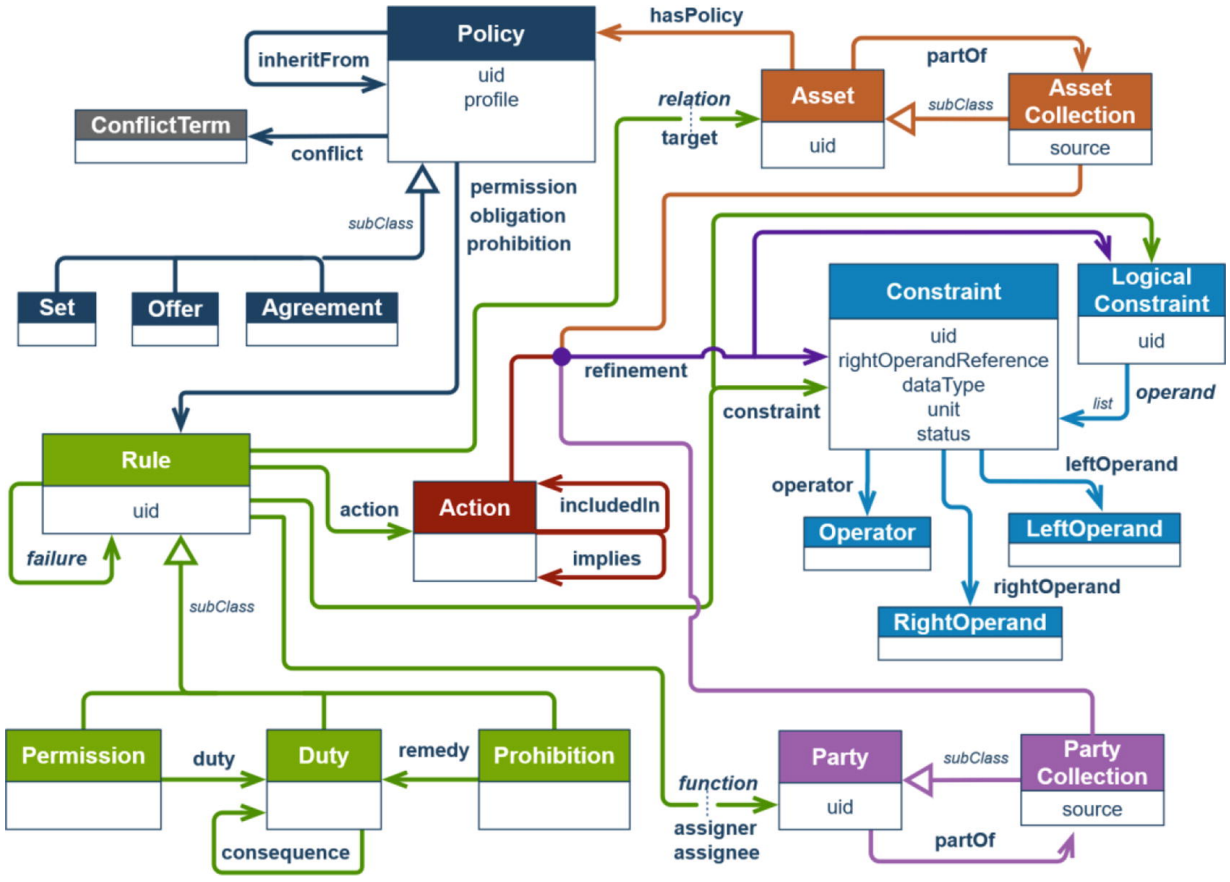


Figure 10-1 W3C ODRL Core Policy Model. Source: W3C ODRL information model, © W3C, used under the W3C Document License

`Assignees` are modeled using the `Agent` pattern and thus may represent roles, teams, or organizations as well as individual users.

ODRL captures governance decisions using the same semantic frameworks used to model domains. These policies can be enforced by

downstream security mechanisms such as graph-level access control, API gateways, or application logic. This separation aligns with the distinction between governance and security: governance defines policy, while security enforces it.

By representing governance policies explicitly, Semantic Knowledge Graphs treat access intent, usage constraints, and accountability as first-class metadata. Policies can be queried alongside data products in a catalog, inspected during design reviews, and referenced during validation and publication workflows. In later chapters, this same policy metadata will be shown to play a role in discovery, quality assessment, and retrieval-augmented generation, reinforcing the idea that governance is most effective when it is embedded directly into the semantic structure of the system.

Data Products and Access Interfaces

Throughout this section, the term data product is used to refer to a managed, governed data asset that is treated as a first-class component of the system rather than a passive dataset. A data product has defined ownership, intended use, and quality expectations. It is published for consumption by other teams or applications. To support controlled access, data products are typically exposed through one or more ports, which represent well-defined interfaces through which data can be queried, retrieved, or observed.

At this point, these terms are used only at a conceptual level to illustrate how governance policies may be applied to concrete system artifacts. A detailed treatment of data products, ports, and their role in Domain-Driven Design and data-mesh architectures is provided in **Chapters 11 and 13**.

Example: A Governance Policy for Accessing a Data Product Port

To make the role of semantic policy models more concrete, consider a simple governance rule applied to a data product port. Suppose a data product exposes an analytic output port intended for internal reporting

and experimentation. Governance requires that this port be accessible only to authorized analytics teams, while remaining invisible to external consumers.

Using the ODRL vocabulary, this rule can be expressed as a policy that grants permission to perform a specific action on a specific asset. The target of the policy is the analytic output port, the assignee is a role representing authorized internal analysts, and the action represents read or query access. Importantly, this policy expresses intent rather than enforcement: it states what should be allowed, not how access is technically controlled.

Conceptually, the policy can be read as

> *Members with the Internal Analytics role are permitted to read data through the Creator Analytics Output Port.*

Represented as semantic metadata, this policy becomes part of the knowledge graph itself. It can be discovered through the data catalog, inspected during governance reviews, and evaluated by downstream systems responsible for enforcing access. Because the policy is expressed independently of any specific security mechanism, it can remain stable even as enforcement technologies evolve.

This example illustrates how governance rules, which are often documented only informally, can instead be captured explicitly using shared semantic vocabularies. By doing so, Semantic Knowledge Graph systems make access intent transparent and machine-interpretable, supporting controlled evolution without sacrificing flexibility.

Governance and Enrichment As Executable Metadata

Governance policy graph models and the reasoners, code, SPARQL, and LLM tools that use them reframe data quality as an outcome of automated processes rather than a subjective label. It is an example of what is often described as governance as code and active metadata—two foundational concepts in Data Fabric architectures (Zaidi & Thoo, 2019).

Figure 10-2 illustrates how governance intent can be embedded directly into a Semantic Knowledge Graph. Policies are modeled as first-class objects linked to data products and their ports. This and all StreamForge examples are from the knowledge graph at https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/knowledge_graph/streamforge_data_catalog_V1.ttl.

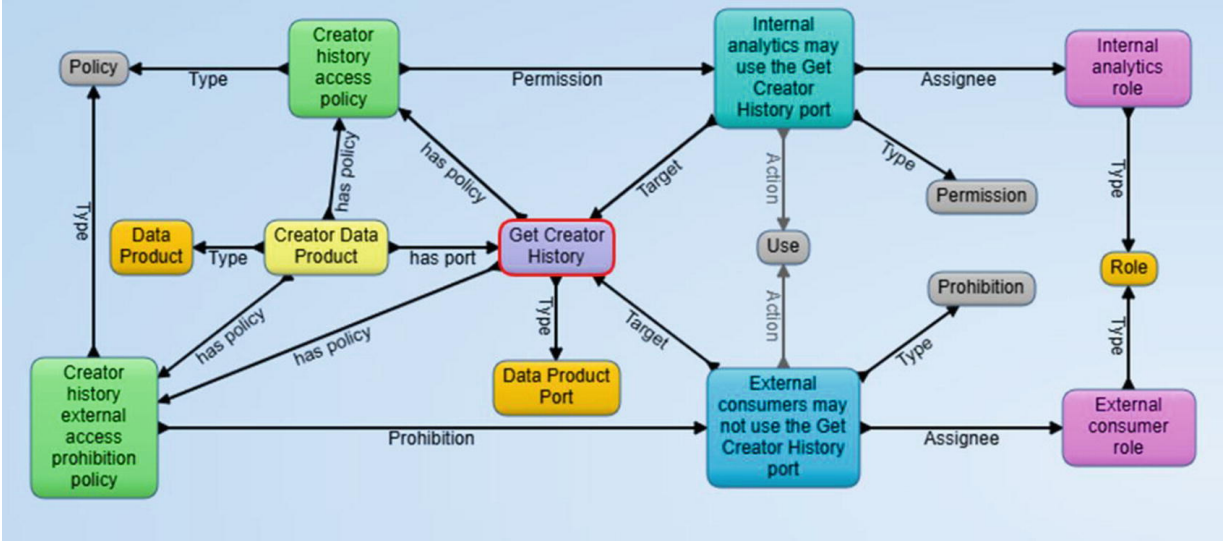


Figure 10-2 Governance policies modeled as semantic knowledge graph metadata using ODRL

Figure 10-2 shows how access intent for a Data Product port can be explicitly modeled. A `Permission Policy` allows internal analytics users to access the `Get_Creator_History` port in the center of the graph, while a complementary `Prohibition Policy` explicitly denies access to external consumers. The policies describe governance intent only; enforcement is handled by downstream security mechanisms.

User-Defined Metadata and Feedback (Collaborative Enrichment)

In addition to automated validation and domain-specific enhancement, data products often accumulate user-defined metadata that reflects real-world usage, perceived quality, and business impact. This includes

annotations (the strings in parentheses are an example of each annotation) such as

- Assessments of usefulness in specific processes ("critical for analyzing user engagement")
- Identification of potential risks ("possible General Data Protection Regulation (GDPR) concerns requiring review")
- Documentation of business value ("essential to $15M in ad revenue last quarter")

Unlike enhancements, which are derived through formal transformation and modeling processes, user-defined metadata is contextual and evolving. It captures operational knowledge that cannot be fully inferred from the data itself.

This form of collaborative enrichment plays a critical role in governance by

- Surfacing trust and risk signals early in the life cycle
- Enabling prioritization of high-value data products
- Providing a feedback loop between data producers and consumers

The last point is especially important for organizations that want to move from a centralized data management process to a process where business units "own" much of the data via data products. These annotations do not alter the underlying data or its semantic model; they provide a separate governance layer that can evolve independently as new knowledge, risks, and usage patterns emerge.

In Semantic Knowledge Graphs, this metadata is represented as structured objects linked to data products and enriched with provenance, policy, and contextual information. Rather than existing as informal notes, governance annotations become queryable, attributable, and actionable components of the graph.

Integrating Governance, Provenance, and Policy

Figure 10-3 shows an example from the StreamForge data catalog of a user-defined governance annotation modeled as entities that integrate provenance, policy, and data product semantics within a single graph.

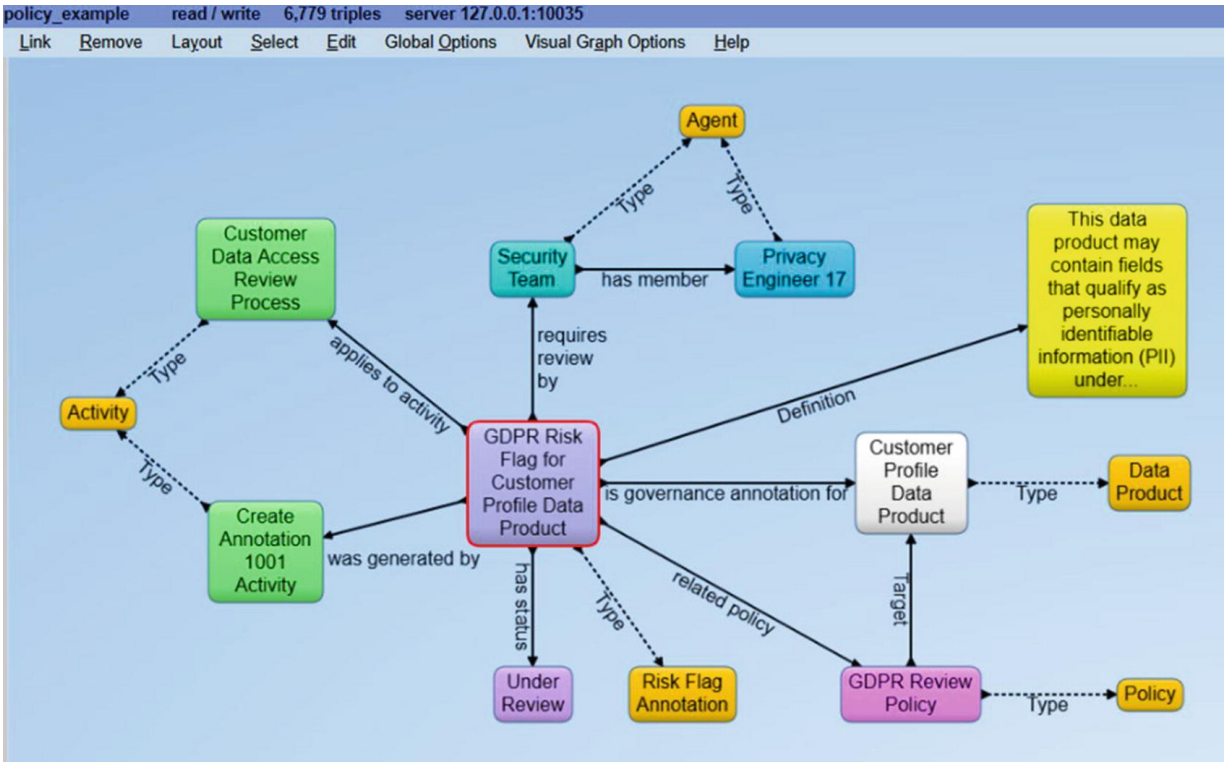


Figure 10-3 Modeling user-defined governance annotations

A `Governance_Annotation` instance (`Risk_Annotation_1001`, of type `Risk_Flag_Annotation`) is linked to the `Customer_Profile_Data_Product`. In this example, a security review (a `prov:Activity`) has identified a possible GDPR issue with the data product.

Provenance is recorded using the PROV-O vocabulary, associating the annotation with the activity that generated it (`prov:wasGeneratedBy`). This allows governance statements to be traced, audited, and evaluated over time.

When governance observations imply required actions, they can be linked to explicit policies modeled using ODRL. In this way, a governance annotation can serve as the trigger or justification for a policy, while the policy itself formally defines permissions, prohibitions, or obligations. For example, a risk flag indicating possible GDPR concerns might be associated with a policy requiring review by a security team before broader access is granted.

Together, these elements demonstrate how Semantic Knowledge Graphs unify descriptive metadata (DCAT and SKOS), provenance (PROV), policy intent (ODRL), and domain-specific extensions (the data product catalog ontology) into a coherent model.

Internal Customers and Quality Metrics

Data products exist to be used. To make this explicit, data products can be linked to their internal customers: the teams, departments, or organizational units that rely on them. Internal customers are modeled as agents, allowing flexibility while keeping the semantics clear.

Feedback from internal customers can be captured through quality metrics, such as customer satisfaction or adoption indicators. Importantly, these metrics are defined with explicit semantics. This avoids ambiguity and supports consistent interpretation across tools, dashboards, and retrieval-augmented generation systems.

By combining roles, validation outcomes, and customer-oriented metrics in a single semantic model, governance and enrichment become observable, testable aspects of the system rather than abstract aspirations. If this all seems familiar, it should. This is the model that industry technical leaders already apply to Microservices, except here applied to data.

Security

Security in Semantic Knowledge Graph systems is closely related to governance but serves a distinct role. Governance *defines* intent. Who should be allowed to access or modify semantic assets and under what conditions? Security mechanisms are responsible for *enforcing* intent. Unlike modeling, inference, or querying, security is not defined by the RDF or OWL specifications themselves. As a result, security is implemented by graph database vendors using a variety of approaches.

At the standards level, RDF and OWL are deliberately silent on security. They define how statements are represented and how meaning is derived, but they do not prescribe how access to those

statements should be controlled. This separation is intentional: security requirements vary widely across organizations and regulatory environments, and embedding a single security model into the core specifications would be impractical. Instead, semantic systems rely on the underlying database platform and code to provide authentication, authorization, and isolation mechanisms.

It is also important to distinguish between securing the knowledge graph itself and modeling security policies for the broader data ecosystem. Access to the graph is enforced by the triplestore or graph database, often integrated with enterprise identity-management systems such as single sign-on, directory services, or role-based access control. However, many of the assets described by the knowledge graph such as relational databases, data lakes, APIs, files, reports, and applications are secured by their own platforms. In those cases, the knowledge graph does not replace the enforcement mechanisms provided by those systems. Instead, it provides a semantic model of policies, roles, responsibilities, classifications, and access requirements so that governance and security rules can be represented consistently across the enterprise.

In practice, most triplestore implementations enforce security at the level of named graphs. Because statements are associated with specific graphs, access control can be expressed in terms of which graphs a user or role is permitted to read or write. This allows different subsets of statements about the same entities to be selectively visible, without duplicating data or redefining identifiers. For example, core domain models may be broadly accessible, while operational metadata, governance annotations, or sensitive attributes are restricted to specific audiences.

Some systems support more fine-grained security models that go beyond graph-level access. These may include statement-level annotations that work at the level of an RDF triple and add security constraints. Such approaches resemble property-level security in other graph models and can be useful in environments with complex regulatory or confidentiality requirements. These mechanisms are

vendor-specific and may increase the cost and complexity of migrating from one triplestore vendor to another.

Security considerations also extend to access and change control. In governed environments, not all users who can query a knowledge graph are permitted to modify it, and not all modifications are treated equally. Separating authoritative assertions from derived or experimental statements (e.g., by writing them to different named subgraphs) provides a natural boundary for enforcing write permissions and review workflows. This reinforces the governance principle that change should be controlled without preventing iteration.

An important practical consideration is that enterprise-grade triplestores typically provide full transactional guarantees. Despite their graph-based data model, these systems support standard ACID transactions, ensuring that updates are atomic, consistent, isolated, and durable. This is particularly important when Semantic Knowledge Graphs are used as part of operational systems, where partial updates or inconsistent states would be unacceptable. Transactional support does not by itself define a security model, but it helps preserve integrity when multiple users, applications, and automated processes are reading from and writing to the graph concurrently.

In summary, security in Semantic Knowledge Graph systems is not defined by RDF or OWL, but by the database platforms and enterprise systems that implement and enforce access control. Triplestores secure the knowledge graph itself through mechanisms such as graph-scoped access control, optional statement-level annotations, and transactional guarantees. At the same time, the knowledge graph can model security policies for assets governed by other systems, making roles, classifications, access requirements, and provenance explicit. This separation of concerns allows security policies to be represented semantically while enforcement remains with the appropriate operational platforms.

Enrichment

Enrichment refers to adding information that was not present in the original source data. This may include inferred relationships, classifications, quality assessments, usage indicators, or regulatory annotations.

Many teams talk about “Bronze/Silver/Gold” data. These terms usually correspond to a quality hierarchy: Bronze (raw) < Silver (cleaned/validated) < Gold (curated). In a Semantic Knowledge Graph, we can do better—defining quality levels whose meaning is grounded in executable checks. A simple cross-domain quality scheme distinguishes between four levels.

Raw Data: Data that has been ingested but has no recorded checks beyond basic syntactic well-formedness. This is not “bad” data. It is simply data in an early life-cycle state. Storing raw data is often essential for traceability, reprocessing, and provenance.

Basic Validation (Domain-Neutral): Data that has passed general semantic and syntactic checks that do not require domain-specific business rules. For Semantic Knowledge Graphs, this typically includes

1. No OWL reasoner inconsistencies
2. No SHACL constraint violations
3. No tooling errors from automated QA checks (e.g., ROBOT, OOPS!, and standards that are specific to the project or enterprise)

This level establishes that the graph is logically coherent and structurally sound without claiming that it satisfies all domain requirements.

Legal/Regulatory Validation: Data that satisfies constraints required by laws, regulations, or organizational compliance policies (e.g., privacy, retention, consent, access restrictions) such as the United States Health Insurance Portability and Accountability Act (HIPAA) or the European Union’s General Data Protection Regulation (GDPR). This category is intentionally separate from domain semantics: data can be logically consistent and still violate regulatory constraints. These requirements

are typically enforced through SHACL constraints, policy models, and audit/provenance rules.

Domain-Specific Enhancement: Data that has been transformed into semantically consistent and decision-ready form through processes such as anonymization, semantic alignment, entity resolution, and the derivation of higher-level metrics. For example, aligning fields such as `cust_id` and `customerID` into a unified ontology enables integrated analysis across business units. Similarly, removing personally identifiable information allows data to be safely used for research while remaining compliant with regulatory requirements.

This scheme clarifies that enrichment is not only about adding data, but also about increasing the validated and trusted state of a data product. The resulting metadata is active: it is generated and maintained by automated processes and can drive operational behaviors such as publication, routing, access control, alerting, and promotion to downstream consumers.

Enrichment reflects usage and assessment. In decentralized environments, the value of a data product is shaped not only by technical correctness but by how well it serves downstream users. Organizations may capture ratings, usage metrics, or qualitative feedback to provide signals that help data consumers decide which data products to use, and data producers decide where to invest their efforts.

Legal and regulatory context is another form of enrichment. Data products may be subject to specific privacy, retention, or compliance requirements. Explicitly modeling which legal regimes apply and what protections are required provides the semantic hooks needed for governance processes and security mechanisms to function correctly.

Other enrichment activities add structural context. Data products may be classified by domain, maturity, intended audience, or operational status. Relationships such as dependency, derivation, or replacement may be asserted or inferred to support impact analysis and change management. In each case, enrichment increases discoverability, interpretability, and responsible use.

Not all enrichment dimensions admit a simple ordering. While some attributes, such as certification status, can be modeled as ordered scales, others are inherently multidimensional.

Because enrichment introduces derived knowledge, provenance is essential. Enriched statements should be distinguishable from authoritative source assertions, and the processes and agents that produced them should be traceable. Recording enrichment results and associating them with explicit provenance metadata allows them to be reviewed, revised, or retracted without destabilizing the underlying graph. The example of adding a risk annotation in Figure 10-3 illustrates how enrichment knowledge is accompanied by provenance metadata.

Enrichment is iterative. As models evolve, validation techniques improve, regulations change, and usage patterns shift, enrichment annotations may need to be updated. Governance defines which forms of enrichment are permitted, how results are evaluated, and when they are treated as authoritative.

Together, these concerns highlight a recurring theme of this book: *Semantic Knowledge Graphs reward early design discipline*. When meaning, metadata, provenance, and policy are treated as first-class elements of the system, many operational challenges become easier to manage. Named graphs, explicit metadata, and declarative models allow concerns such as access control, validation, and derived knowledge to be addressed without entangling them with core domain semantics.

With this chapter, **Part 1** of the book comes to a close. The chapters in this part have focused on foundational tools, processes, and design principles: how semantic models are created, how data is integrated, how knowledge is validated, and how governance concerns are embedded into the life cycle. The emphasis has been on establishing a common conceptual and technical foundation, independent of any particular architectural style.

Part 2 builds on this foundation by exploring how these principles are applied in larger-scale organizational and architectural contexts.

Topics such as data fabrics, data mesh, and data catalogs introduce new ways of structuring responsibility, ownership, and integration across teams. Semiformal models such as taxonomies and lexical vocabularies provide an entry point and foundation for developing ontologies.

The transition from **Part 1** to **Part 2** reflects a shift in perspective: from individual tools and life-cycle activities to system-level architectures and organizational patterns. With the foundation of the specific modeling languages and tools, we are ready to turn to the architectural choices that shape how Semantic Knowledge Graphs are adopted and scaled in enterprise systems that manage Big Data.

Summary

- *Governance and Security as Life-Cycle Concerns*: Governance and security are not external constraints applied after the fact, but integral aspects of the data life cycle. They must be designed alongside modeling, validation, and enrichment rather than treated as separate compliance activities.
- *Governance in Distributed Architectures*: In distributed environments, governance doesn't rely on centralized control alone. Instead, it can be expressed in ways that allow teams to operate autonomously while still adhering to shared policies, standards, and accountability structures. The degree to which governance is centralized vs. distributed will vary greatly for each industry and organization.
- *Data Quality Levels As Executable Governance Metadata*: Data quality is made explicit through a small, cross-domain set of quality levels that correspond to concrete validation states. These levels provide a shared vocabulary for describing the readiness and trustworthiness of data products without embedding domain-specific assumptions.
- *Derived, Not Asserted, Quality Outcomes*: Quality levels are primarily derived outcomes of automated validation steps. This ensures that quality indicators reflect the current validation state of the data. However, textual quality annotations and issues are also essential

because no enterprise system can predict and automate all quality metrics ahead of time.

- *Governance Roles and Accountability*: Governance responsibilities are modeled explicitly as roles played by agents in relation to data products. This allows accountability, stewardship, and decision-making responsibilities to be represented without hard-coding organizational structure into the ontology.
- *Internal Customers and Quality Metrics*: Data products are linked to their internal customers, making consumption and feedback explicit. Quality metrics, such as customer satisfaction or adoption indicators, complement validation-based quality levels by capturing how data products are perceived and used in practice.
- *Governance and Enrichment As Executable Metadata*: By representing roles, policies, validation checks, and outcomes directly in a Semantic Knowledge Graph, governance becomes executable and observable. This supports automated enforcement, monitoring, and continuous improvement rather than static documentation. It is a key to rewarding early design discipline by reducing future implementation, maintenance, and management effort and costs.
- *Security As Enforced Policy, Not Just Access Control*: Security is treated as a concrete expression of governance, encompassing access control, compliance constraints, and policy enforcement. Modeling security requirements explicitly enables them to be validated, audited, and reasoned over alongside other governance concerns.
- *Enrichment Beyond Adding Facts*: Enrichment includes not only adding new data but also increasing trust and usability through validation, provenance, and governance signals. Advancing a data product through validation levels is itself a form of enrichment.
- *Foundation for Scalable, Governed Systems*: Together, these patterns show how Semantic Knowledge Graphs provide a foundation for scalable, governed data systems in which quality, accountability, and security are first-class, machine-interpretable concerns rather than informal agreements.

- *Further Reading*: For more information on data enrichment and managing enterprise data in general, see *Data Management at Scale* (Strengholt, 2023). For a deep dive into governance, see *Data Governance: The Definitive Guide* (Eryurek et al., 2021).
- *Transition from basic tool topics to application considerations*. This completes **Part 1** of the book, which has covered the basic tools that all Semantic Knowledge Graph developers need to understand. To use an analogy with carpentry, this first section covers the saws, hammers, and nails. The next part of this book describes specific architectures and applications. To continue the analogy, we now turn to the different skills required to build a cathedral vs. a skyscraper.

Part 2
Enterprise Architectures and Organizational Patterns

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_11

11. Enterprise Data Products: Microservices for Data

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

Over the past decade, *microservices* have changed how enterprises build and govern software. Systems based on microservices are

- Decomposed into loosely coupled services with explicit interfaces. This facilitates reuse and rapid change.
- Independently deployable and aligned to business domains rather than a central IT organization.

This shift enables continuous evolution, CI/CD, improved resilience, distributed ownership, and most importantly agile systems that can rapidly change to exploit new business opportunities.

In many organizations, software is no longer a tool to help run the business; software ***is*** the business. For companies such as Amazon, Google, and LinkedIn, software is more important than buildings or infrastructure. This new way of doing business required a new model for software development that facilitates rapid, agile change. Domain teams now build and operate the systems that define marketing analytics, logistics networks, pricing engines, and customer platforms. Governance moved from centralized control to shared standards with local autonomy.

But this transformation applied primarily to code, not to data.

While application logic became modular and domain-aligned, data remained centralized in warehouses or lakes, managed by specialized teams, and exposed through brittle pipelines. The result was an architectural imbalance: decentralized agile services built on centralized, slow-moving data platforms.

The concept of the *data product* addresses this imbalance. A data product applies microservice principles to data. Data is treated as a first-class asset designed for consumption, not as exhaust from operational systems.

Data products extend the logic of modularity and contract-based interaction from services to the data they produce and consume. This is the natural next step in the evolution from monolithic systems to distributed, loosely coupled, domain-driven architectures. To make this more precise, we now define in detail the concept of a data product.

What Is a Data Product?

A data product is a domain-owned, self-contained unit that is designed for consumption by other systems and users. It exposes data through well-defined, versioned interfaces called ports. Users are often internal to the organization as well as customers and external users.

Like a microservice, a data product is

- Owned by a specific domain team
- Independently deployable and evolvable
- Accessed through explicit contracts (APIs, event streams, or queries)
- Responsible for its own quality, documentation, and governance

The key distinction is that a microservice exposes behavior, while a data product exposes data.

In Semantic Knowledge Graph architectures, the semantic model is part of the data product itself, not external documentation. Ontologies, vocabularies, and metadata define the data product, ensuring that meaning is explicit, shareable, and enforceable across systems.

A more detailed treatment of data products, including their relationship to Domain-Driven Design (DDD), is provided in **Chapter 13**.

Integrating Operational and Analytic Data

Traditional architectures separate operational and analytic concerns:

- Operational systems, also known as Online Transaction Processing (OLTP) systems, focus on transactions, consistency, and real-time processing.
- Analytic systems focus on aggregation, reporting, and historical analysis.

This separation introduced pipelines, duplication, and delays between data creation and data use. Data products reduce this gap by bringing operational and analytic perspectives closer together. A well-designed data product often

- Publishes real-time events derived from operational systems
- Maintains structured datasets for analysis and reporting
- Provides semantic metadata and context for interpretation
- Supports both machine consumption (APIs, streams) and human consumption (queries, dashboards)

In this role, a data product acts as a bridge:

- Between systems of record and analytic systems. Enterprise data is often copied into data lakes, caches, reports, and analytic systems. The system of record is the authoritative operational source treated as definitive for the current state of that data.
- Between event streams and analytic models.
- Between raw data and interpretable, governed information.

This integration reduces latency, duplication, and ambiguity, enabling a shift from batch-oriented pipelines to *continuous, domain-driven data flows*.

A concrete example of this pattern appears in the StreamForge reference architecture used throughout this book. Recall that

StreamForge is a hypothetical video streaming company. For example, the `SF_Creator_Insights_Data_Product` combines operational event streams (such as user engagement events) with curated analytic datasets to produce metrics, KPIs, and derived insights. These outputs are exposed through well-defined interfaces called ports and enriched with semantic metadata, making them reusable across domains without requiring downstream teams to reconstruct context or meaning.

Data Mesh and Data Fabric: Complementary, Not Competing

Much of the current discussion frames Data Mesh and Data Fabric as competing architectural choices. They are better understood as related responses to the same underlying problem: how to scale data management in large, complex organizations without recreating the bottlenecks of centralized data platforms. Both approaches emerged as reactions against earlier models such as enterprise data warehouses and data lakes that promised global integration but often became slow-moving, brittle, and disconnected from the needs of individual teams. As data volumes, data sources, and organizational complexity increased, these centralized approaches struggled to keep pace. The main difference is that the two models focus on different aspects of the problem:

- Data Mesh addresses who *owns what* and how *responsibility* is distributed.
- Data Mesh focuses on organizational alignment: domain ownership, federated governance, and treating data as a product.
- Data Fabric addresses how data is *connected, governed, and operationalized* across that distribution.
- Data Fabric focuses on integration and automation: shared infrastructure, metadata, lineage, and policy enforcement across distributed systems.

Shared Principles and Capabilities

Data Mesh and Data Fabric share more similarities than differences. Both Data Mesh and Data Fabric emphasize

- *Metadata as a first-class concern*: Metadata is not an afterthought documentation exercise. It is a core mechanism for discovery, governance, interoperability, and automation. Technical metadata (schemas, formats, lineage), business metadata (definitions, ownership), and semantic metadata (conceptual models and vocabularies) are essential to making data usable at scale.
- *Self-service access to data*: Both architectures aim to reduce reliance on centralized data engineering teams for routine access and integration. Users should be able to discover, understand, and use data products without needing hand-crafted integrations for every new use case.
- *Data products as the unit of reuse*: Rather than treating datasets as raw exhaust from operational systems, both approaches promote the idea of well-defined, documented, versioned, and governed assets (i.e., data products) designed for consumption by others.
- *Loose coupling and composability*: Data producers and consumers should evolve independently. Interfaces, schemas, and contracts matter more than physical storage or internal implementation details. This decoupling reduces coordination costs and enables parallel development across teams. The most common way to deliver loose coupling is via the use of a publish/subscribe message or event middleware bus such as RabbitMQ or Kafka.
- *Semantic models for shared understanding*: Both approaches recognize that technical schemas alone are insufficient. Shared meaning, captured through vocabularies, taxonomies, and ontologies, is necessary for interoperability across domains.

Hopefully, the reader can see, based on what we've described so far, how Semantic Knowledge Graph technology (OWL, SHACL, and SPARQL) combined with Graph RAG is ideally suited to model, validate, reason over, and discover data products.

Where They Differ: Governance and Control

The primary distinction between Data Mesh and Data Fabric lies in how governance is organized, not in what governance exists.

Data Mesh Emphasizes Distributed Governance

Ownership of data products is pushed to domain teams who are closest to the data and its business meaning. Governance is implemented through shared standards, templates, and platform capabilities rather than centralized approval processes. The goal is to scale data management by aligning it with domain expertise and existing organizational structures.

This approach is particularly attractive in organizations that already operate with a high degree of autonomy across teams, such as large technology companies, digital-native organizations, and enterprises that have successfully adopted microservices.

Data Fabric Emphasizes Centralized Governance and Coordination

While it still supports self-service access and automation, a Data Fabric typically relies on a more centralized layer that enforces standards, manages metadata, and provides integrated views across the enterprise. Governance policies are often defined and applied centrally, with tooling used to automate compliance and consistency. This model is common in highly regulated industries such as healthcare and government, where consistency, auditability, and risk management are paramount.

A Spectrum, Not a Binary Choice

Data Mesh and Data Fabric are not mutually exclusive. Organizations operate somewhere along a continuum between the two.

A company may adopt Data Mesh principles for domain ownership and data product design, while relying on a centralized Data Fabric for metadata management, lineage tracking, and policy enforcement. Another organization may begin with a Data Fabric to establish

baseline governance and tooling, then progressively decentralize ownership as teams mature. Even within a single enterprise, different domains may sit at different points on the spectrum depending on regulatory requirements, risk tolerance, and organizational maturity.

The important lesson is that architecture follows organization. This is an example of Conway's Law (Conway, 1999), which states that systems tend to mirror the communication structures of the organizations that build them. Just as the transition from monolithic systems to microservices required changes in team structure, ownership, and incentives, successful data architectures must align with how an organization actually operates.

The Role of Semantic Knowledge Graphs

Semantic Knowledge Graphs (SKGs) play a critical role here by providing a shared semantic layer that handles both sets of considerations via explicit knowledge representation and automated reasoning.

SKGs enable data products to be

- Semantically explicit rather than schema-only
- Integrated across systems without tight coupling
- Governed through shared vocabularies rather than hard-coded assumptions

Semantic Knowledge Graphs provide a unifying semantic layer that works equally well in both Data Mesh and Data Fabric contexts. In a Data Mesh, SKGs help maintain coherence across independently developed data products by providing shared vocabularies and explicit semantic mappings. In a Data Fabric, they support centralized discovery, lineage, impact analysis, and governance automation.

In both cases, SKGs act as the connective tissue between distributed data assets, enabling interoperability without requiring uniform physical storage or rigid schema alignment.

A Reference Architecture for Enterprise Data

Figure 11-1 presents a conceptual reference architecture that situates Semantic Knowledge Graphs within the broader ecosystem of enterprise data systems. It provides a shared model for understanding how data products, semantic models, governance, analytics, and AI services coexist in practice. The technologies shown are illustrative choices, not a required stack. The discussion below also considers supporting tools, such as Docker and Airflow, that are not shown separately in the figure.

One of the reasons discussions of Data Mesh and Data Fabric often become polarized is that they are framed as architectural alternatives. In reality, most successful implementations rely on largely the same technical components. The differences lie primarily in how ownership, governance, and control are organized across the enterprise.

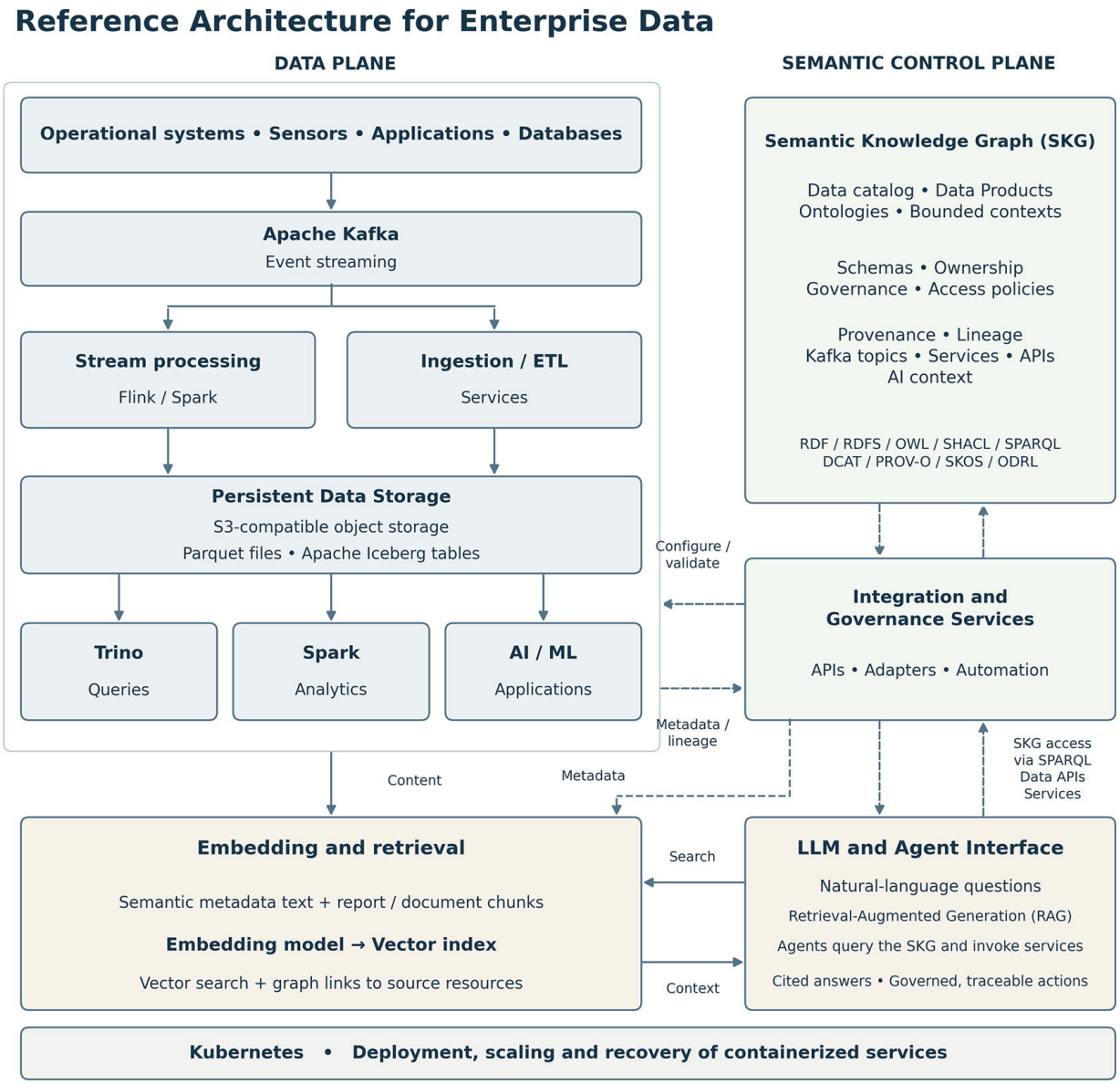


Figure 11-1 Reference architecture for enterprise data. The semantic control plane manages meaning, metadata, and governance; Kubernetes manages deployment, scaling, and runtime lifecycle. The technologies shown are illustrative choices supporting overlapping approaches such as Data Lake, Lakehouse, Data Mesh, Data Fabric, and Knowledge Management

The term plane describes a major architectural area of responsibility, such as a control plane, platform plane, or data product experience plane. I use the term in the same broad sense as Dehghani (Dehghani, 2022). A plane is not a single component, but a coordinated layer of services, interfaces, policies, and responsibilities that supports part of the overall data architecture.

In Figure 11-1, the data plane carries, processes, stores, and serves enterprise data. The semantic control plane uses a Semantic Knowledge Graph to represent meaning, ownership, contracts, governance policies, and lineage. Integration and governance services connect these descriptions to operational behavior. These are logical responsibilities: they do not require all data or metadata to reside in one physical store, or all governance decisions to be made centrally.

Data products connect the two planes: their datasets, event streams, and services expose explicit ports, while the graph describes their semantics, ownership, and dependencies. Persistent Data Storage represents the storage role; it does not commit the architecture to a particular organizational approach such as Data Mesh or Data Fabric.

At the bottom of the figure, text derived from semantic metadata and document chunks supports vector embedding and retrieval. The LLM interface can use retrieved context and access the Semantic Knowledge Graph and services through explicit interfaces. Kubernetes supports the runtime deployment of containerized services across the architecture. Its control plane manages infrastructure; it is distinct from the semantic control plane. **Chapter 13** develops the semantic data catalog and Graph RAG portions of this architecture.

Event and Messaging Backbone

The foundation for both Data Mesh and Data Fabric architectures is a publish/subscribe messaging backbone that enables loose coupling between services and data products. Rather than integrating systems through point-to-point connections, data producers publish events/messages that can be consumed by multiple downstream systems.

Message buses have been used to enable loose coupling in distributed systems since the late 1990s. By the early 2000s, they were a best practice most large enterprises adopted. These systems focused on reliable message delivery, treating messages as transient units of communication between producers and consumers. Over the past

decade, however, a shift has occurred toward event-driven architectures.

This shift has been driven by dramatic reductions in the cost of storage and the maturation of distributed systems: what once required expensive, specialized infrastructure can now be achieved with commodity hardware and cloud services at a fraction of the cost. As a result, modern event buses (exemplified by platforms such as Apache Kafka) treat events as immutable records of state changes that are durably stored and can be replayed. This reflects a deeper conceptual change: from sending messages that are consumed and discarded to *maintaining persistent streams of events* that can be independently processed, audited, and reused across multiple systems.

This event-driven approach provides scalability, resilience, and independent evolution of producers and consumers. It aligns naturally with the concept of data products as autonomous units that expose well-defined interfaces.

Bus architectures are fundamentally about loose coupling. In synchronous systems, failures propagate: when one service becomes unavailable, dependent services block or fail in turn. A bus breaks this dependency chain by persisting messages/events independently of consumers, allowing systems to process information asynchronously and *recover without cascading failure*.

Modern event platforms such as Kafka take this a step further. The bus is no longer about exchanging messages between services but rather recording business events. This enables architectures based on persistent, replicated event logs that support massive parallelism, high throughput, and strong fault tolerance. Messages are *transient communications*. Events are *durable facts* that can be replayed, reprocessed, and analyzed over time.

Equally important are the business implications. Event-driven architectures encourage systems to be modeled around business events rather than technical messages. Events such as placing an order, watching a video, issuing an invoice, or admitting a patient correspond directly to how organizations understand and manage their operations.

When these events are modeled explicitly using ontologies to define their meaning and relationships, the resulting systems become more understandable, more maintainable, and easier to evolve.

This shift also aligns with broader organizational trends. Making business events first-class architectural elements shifts control away from centralized IT toward domain teams who understand the semantics of their own processes. Ontologies play a critical role here, providing a shared, formal vocabulary that allows independently developed systems to interoperate while remaining loosely coupled.

Service Packaging and Deployment

Data products and supporting services must be packaged in a way that allows them to be deployed, scaled, and updated independently. Modern architectures typically achieve this through lightweight, container-based packaging.

Containerization allows teams to bundle application logic, data, configuration, and dependencies into self-contained units that can be deployed consistently across environments. It also provides a natural boundary around data products, reinforcing the idea that each product is owned, versioned, and operated independently.

A widely used container tool is Docker, but again, the key point is not the tool itself. The important idea is that data products are treated as deployable services rather than as passive datasets.

Orchestration, Scaling, and Discovery

Once services are packaged, they must be orchestrated: deployed across infrastructure, scaled based on demand, monitored for health, and made discoverable to other services. This orchestration layer plays a critical role in enabling self-service access while maintaining operational stability.

In practice, orchestration platforms also provide service discovery mechanisms that allow consumers to locate and interact with data products without hard-coded knowledge of deployment details. This is essential for supporting the loose coupling emphasized by both Data

Mesh and Data Fabric. A common example is Kubernetes, though the architectural role can be fulfilled by a variety of platforms, including managed cloud services.

Semantic and Vector Data Stores

Semantic Knowledge Graphs occupy a central position in this reference architecture. They provide a shared semantic layer that supports metadata management, governance, lineage, reasoning, and increasingly, vector-based retrieval.

In both Data Mesh and Data Fabric contexts, the knowledge graph acts as a unifying substrate rather than a centralized data store. It connects independently owned data products through shared meaning, enabling discovery, interoperability, and policy enforcement without requiring uniform physical schemas or storage technologies.

All of the major triplestore graph database vendors now offer combined semantic graph capabilities with vector support. Semantics and metadata are treated as first-class assets, not as documentation artifacts.

Relational Data Stores

Despite the rise of graph databases and specialized analytic platforms, relational systems remain the backbone of enterprise data architecture. Core business domains such as customer records, financial transactions, and clickstream management continue to be modeled and managed most effectively using relational databases.

Semantic Knowledge Graphs do not eliminate the need for relational or other data stores; rather, they provide a unifying semantic layer that integrates these systems into a coherent whole. In the architectures described in this book, relational databases are treated as first-class components: authoritative sources of record, key inputs to data products, and essential contributors to enterprise data ecosystems.

The influence of the relational model also extends far beyond traditional transactional systems. Modern data platforms such as

Google BigQuery rely on SQL and relational abstractions to query and process massive datasets. Streaming architectures also use the relational model for more than storing business data. For example, Confluent's ksqlDB enables developers to execute continuous SQL queries over Kafka event streams, supporting joins, aggregations, and transformations in real time.

Relational systems are integrated as peers within modern architectures such as Data Mesh and Data Fabric. They act as systems of record, producers and consumers of event streams, and foundational components for both operational and analytic workloads. The enduring dominance of SQL, across transactional, analytic, and streaming contexts, demonstrates that relational paradigms remain central and long-lived, while Semantic Knowledge Graphs add valuable new capabilities for integration, governance, and meaning.

Data Pipelines and Workflow Orchestration

Ingestion, transformation, enrichment, and synchronization are managed by explicit data pipelines. These pipelines connect event streams, operational systems, analytic platforms, and knowledge graphs. They ensure that data products are continuously updated and governed. Workflow orchestration tools such as Apache Airflow and Prefect provide scheduling, dependency management, and monitoring for these pipelines.

These tools operate at a different level than infrastructure orchestrators such as Kubernetes, which manage how services are deployed and scaled. Workflow orchestrators focus instead on *what* data processing tasks should run and *when*, rather than where and how they are executed.

Language Models and AI Services

Increasingly, enterprise architectures include large language models and other AI services that interact with data products. These models support use cases such as natural-language querying, semantic enrichment, classification, and retrieval-augmented generation.

In this reference architecture, language models are treated as consumers and producers of data products, rather than as monolithic systems*. Semantic Knowledge Graphs play a critical role in grounding model behavior, managing context, and reducing the risk of hallucination*.

Interface Definition and Contracts

Finally, data products require explicit contracts that define how they are accessed and consumed. These contracts specify available operations, schemas, and constraints, enabling consumers to integrate without direct knowledge of internal implementations.

Interface Definition Languages (IDLs) provide a formal mechanism for expressing these contracts. In the context of Data Mesh and Data Fabric, they support discoverability, versioning, and independent evolution of data products. Examples include GraphQL's schema definition language (graphql.org, 2026), OpenAPI descriptions for REST APIs (OpenAPI Initiative, 2026), and Protocol Buffers service definitions for gRPC (Zhuang & Anderson, 2024).

Sidecars: Executing Governance Alongside the Data Product

Data Mesh distributes ownership to domain teams while maintaining consistent governance. The sidecar architecture provides the model to support this.

A sidecar is a process that runs alongside a data product and shares its execution context. The sidecar is responsible only for cross-cutting concerns such as policy enforcement, observability, and discovery. Governance logic is not embedded in domain code; it is executed next to it.

The distinction is simple but important. Data products are assets that are cataloged, versioned, and consumed. Sidecars are runtime processes that enforce standardized behavior. Keeping governance execution separate from domain logic prevents drift toward either centralized bottlenecks or unenforced policies.

In practice, sidecars are often implemented using container-based deployments (e.g., separate containers sharing the same runtime environment). However, the technology is secondary. The architectural principle is what matters: governance rules are defined centrally and enforced automatically at run time.

Sidecars reinforce a broader lesson of Data Mesh and Data Fabric architectures: governance is not something you document. *It is something you run*. Figure 11-2 illustrates one deployment pattern for a data product and its governance sidecar within the enterprise reference architecture shown in Figure 11-1.

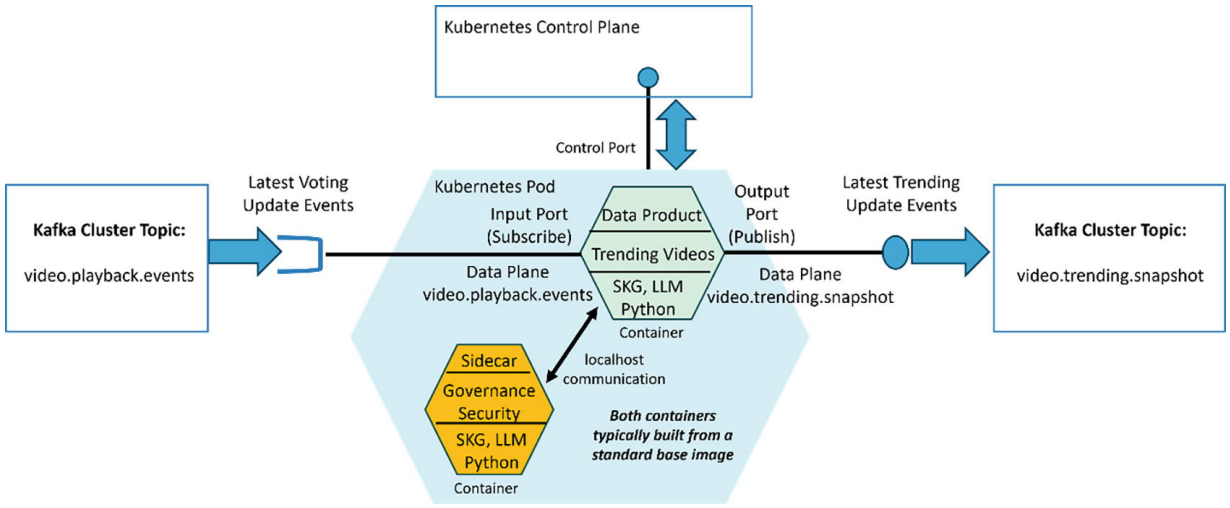


Figure 11-2 Deployment view of a data product with governance sidecar

A data product can be deployed as a Kubernetes Pod containing two containers: the data product itself and a governance sidecar. The product exposes explicit input and output ports that connect to Kafka topics via the data plane, while Kubernetes manages lifecycle and scaling through the Kubernetes control plane.

The sidecar communicates locally with the data product, enforcing governance policies without embedding cross-cutting logic in domain code.

The diagram makes the separation of concerns explicit. The data product defines *contractual interfaces*: input ports subscribe to topics. A topic is a named event stream for specific kinds of events such as

playing a video or making a purchase that a microservice or data product can subscribe and/or publish to. An output port publishes derived data to downstream consumer topics. Consumer in this context means a system that consumes events published to the bus, not necessarily a customer. These ports represent stable interfaces independent of the underlying bus technology.

At run time, the data product and its governance sidecar run in separate containers within the same Kubernetes Pod. They share a network namespace and can communicate through localhost, while maintaining a clean separation between domain logic and cross-cutting concerns such as access control, observability, lineage, and compliance.

The distinction between the data plane and the Kubernetes control plane is equally important. The bus carries domain events between independently evolving systems, while Kubernetes manages deployment, scaling, and health. The semantic control plane shown in Figure 11-1 provides meaning, metadata, and governance policies; services such as the sidecar apply those policies at run time.

Why This Architecture Matters

This conceptual reference architecture highlights an important point: Data Mesh and Data Fabric are not defined by their technology stacks. They are organizational and governance patterns built on a largely shared set of technical capabilities.

Semantic Knowledge Graphs are uniquely positioned within this architecture because they support both models simultaneously. They enable federation without fragmentation and centralization without rigidity. As the following chapters will show, this makes them a foundational component of scalable, enterprise-grade data systems.

Looking Ahead

The remainder of **Part 2** explores these ideas in more detail. We examine how data products are designed and governed, how event-driven architectures support loosely coupled integration, and how

Semantic Knowledge Graphs enable scalable, federated data ecosystems. Rather than advocating a single "correct" architecture, the focus is on understanding the trade-offs and designing systems that can evolve as organizations and requirements change.

The shift from centralized to federated data architectures mirrors the earlier transition from monoliths to microservices. The lesson from that transition is clear: *technical architecture and organizational structure evolve together*. Understanding that relationship is essential for designing agile data systems that can rapidly evolve with the enterprise rather than constrain it.

Summary

- *Microservices transformed enterprise software by decentralizing ownership*, enabling loosely coupled services with explicit interfaces aligned to business domains—but this shift largely excluded data.
- *Traditional centralized data platforms (warehouses, lakes, catalogs, and lakehouses) created an architectural imbalance when paired with decentralized application services.*
- *The data product concept applies microservice principles to data*: domain ownership, explicit contracts, versioning, discoverability, and built-in governance.
- *Data Mesh emphasizes distributed domain ownership and federated governance*. Data Fabric emphasizes centralized coordination, metadata management, integration, and policy enforcement. However, they define continuums rather than discrete, nonoverlapping concepts. In reality, few if any organizations will have completely federated or completely centralized governance.
- *Both approaches share core principles:* metadata as a first-class concern, self-service access, loosely coupled and composable data products, and the need for shared semantic models. The key distinction between Mesh and Fabric lies in governance organization (distributed vs. centralized), and most enterprises operate along a spectrum between the two.

- *Semantic Knowledge Graphs (SKGs) provide a unifying semantic layer that supports and unifies Mesh and Fabric* by enabling interoperability, shared meaning, lineage, and governance across distributed systems.
- *A conceptual reference architecture illustrates common building blocks of modern enterprise data systems:* event-driven backbones, containerized service packaging, orchestration layers, semantic and relational stores, analytic platforms, pipelines, AI services, and interface contracts.
- *Event-driven architectures promote loose coupling, resilience, and alignment with business events*, with ontologies providing formal definitions of event semantics.
- *Governance is not a separate layer but an overlay across all components*, implemented through metadata, policies, and automated enforcement.
- *Sidecar patterns allow governance logic to run alongside data products without embedding it directly in domain code*, balancing autonomy with enforceable standards.
- *Technical architecture and organizational structure evolve together*; successful data architectures align with how enterprises actually operate rather than with abstract models.
- *Further Reading*: *Microservices Overview* (Fowler & Lewis, 2014), *Data Mesh Overview* (Dehghani, 2020), *Data Mesh Book* (Dehghani, 2022), *Gartner on Data Fabrics* (Zaidi & Thoo, 2019), *Enterprise Data Management with Data Fabric and Data Mesh* (Strengholt, 2023).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_12

12. Semiformal Modeling

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

> *Derived from the workflows of librarians, the Ontology Pipeline® offers a systematic methodology for constructing semantic knowledge management systems. The pipeline consists of iterative building blocks, with each phase preparing for the next stage of building.*
> —Jessica Talisman (Talisman, 2025)

This chapter introduces a family of structures that are essential to Semantic Knowledge Graphs but are not formal ontologies. I call these *semiformal* models. The Semantic Web languages such as OWL and SHACL are Very High Level Languages (VHLL). The goal of a VHLL is to provide languages that can model data and process in a way that a computer can execute while being easier for nontechnical people to understand than a programming language. Such languages go back to the first AI boom in the 1980s fueled by expert systems. The expert systems languages were primarily if-then rules with inference engines that could do forward and backward chaining. The Semantic Web standards include rules but also provide VHLLs that go beyond rules such as OWL and SHACL.

In contrast to a VHLL, a semiformal model is concerned with defining models that are intuitive to users but typically not executable. A good example that we will explore in detail in this chapter is an organization chart. An organization chart summarizes the structure of a corporation. Typically, it doesn't follow any particular formal model. Some links may represent subclass relations, others instance relations, and others whole-part component models. Semiformal models are defined using techniques from

Information Architecture (IA) that emphasize user understanding over formal models.

Understanding semiformal models is crucial because many technologists, including experienced ontologists, initially underestimate their value. For a long time, I struggled to understand the practice of IA because the structures it produced did not correspond to any single formal relationship like subclassing or part–whole decomposition. IA practitioners intuitively mix multiple organizing principles such as geography, products, processes, markets, topics, life-cycle stages, in order to convey meaning, while I kept trying to force everything into one clean hierarchy. When I spoke with IA experts, they never explicitly mentioned the requirement to make things as intuitive to nontechnical users as possible because, to them, it is obvious. To those of us trained to think in terms of logic and sets, it is not.

The clearest article I've seen that explains the synergy between these semiformal models and ontologies is Jessica Talisman's Ontology Pipeline framework (Talisman, 2025). She explains that semiformal models are not primitive precursors that get replaced; they are foundational components of any scalable semantic ecosystem. Her framework—controlled vocabulary ➤ metadata standards ➤ taxonomy ➤ thesaurus ➤ ontology ➤ knowledge graph—shows that well-structured semiformal models enable effective ontology design rather than compete with it. This chapter is significantly based on her ideas, and I encourage all readers to read her paper. Organizations that attempt to build ontologies without these earlier stages often struggle with messy terminology, unclear boundaries, inconsistent categories, and a lack of shared understanding. Pre-existing semiformal models offer very useful input data to an ontology developer, while the creation of new ones can be a significant aid in communicating with subject matter experts as well as a first step toward an ontology.

This chapter examines the major types of semiformal models that support both information architecture and semantic design. We will focus on five models emphasized in Talisman's work: glossaries, controlled lexical vocabularies, metadata standards, taxonomies, and thesauri and a sixth model, the folksonomy, which emerges from user behavior rather than formal governance. Together, these structures bridge informal human meaning and formal logical semantics.

Figure 12-1 shows various types of models and maps them in two dimensions:

1. The X axis measures how intuitive a model is for a domain expert who is not a developer.
2. The Y axis measures explicit knowledge representation that can be utilized by computers for automated reasoning. Note that the Y axis is not just about being machine-interpretable. It is about being both machine-interpretable and providing the ability to model at levels that abstract away from code, i.e., a VHLL.

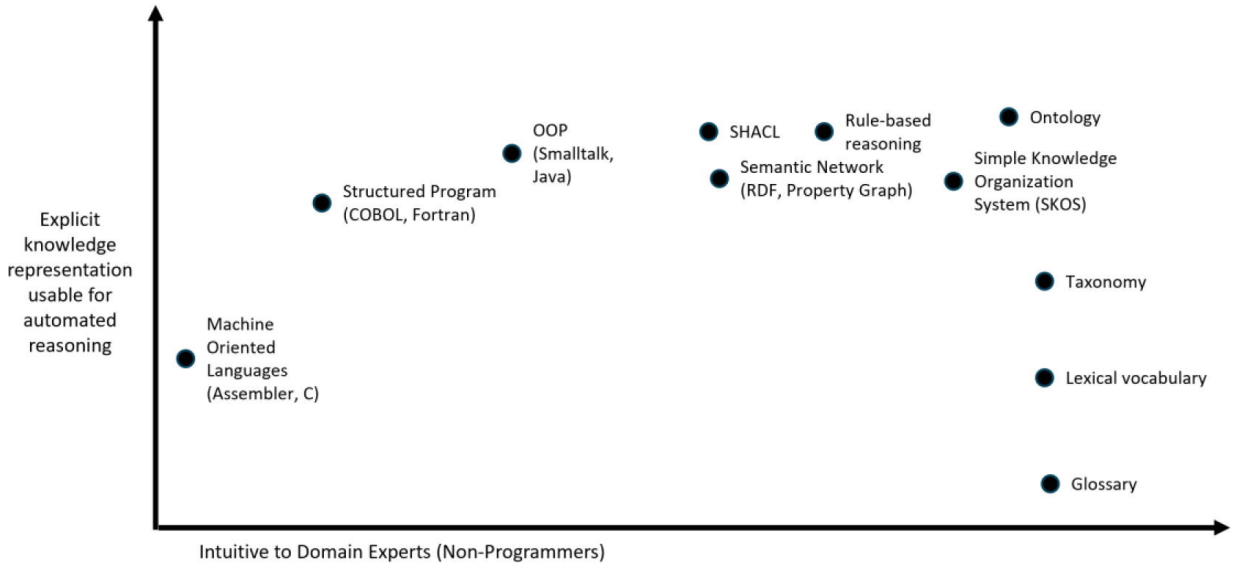


Figure 12-1 Modeling for human and machine interpretation

The graph is meant to be conceptual. There are no specific metrics I can point to, and one could argue that some data points should be higher/lower or further to the left/right. The point of the graph is not to provide a rigid meta-model that graphs each type of model but rather to communicate the concept that OWL represents the convergence of two different trends in software development:

1. Symbolic AI and software engineering research has been creating VHLLs that decrease the gap between developers and domain experts.
2. Information Architecture has developed various models to intuitively standardize the description and definition of domains in order to provide intuitive structure to web sites, menus, and diagrams.

In the earliest days of software development, developers were confined to languages such as Assembler that mapped very closely to the machine languages hard coded into CPUs. These languages didn't have basic concepts such as strings or base 10 numbers. Early languages required manual memory management and binary arithmetic.

These languages were extremely difficult to develop and maintain. This soon gave way to higher-level languages such as COBOL for business programming and Fortran for scientific programming. Developers established heuristics for well-structured programs such as never using the "go to" option that allowed a program to jump to any random place in the code. The next advance was Object-Oriented Programming (OOP) languages such as Smalltalk and Java that allowed developers to define code in terms of business objects such as Account and Customer.

At about the same time as OOP migrated from research to industry, rule-based systems migrated from AI labs to industry and allowed developers to define business logic in VHLLs based on Rules and forward/backward chaining. Rules were a dramatic advance to define business logic. Expert system vendors would often claim that their tools were so intuitive that domain experts could use them directly. While that may have been true for a small number of "power user" domain experts, developers were still required for large-scale problems and integration with other systems and databases.

All of these models are what software engineers call "formal models." This term is sometimes reserved for VHLLs based on logic, but it is also used to distinguish any high-level language that can be executed by a digital computer, which is the way I use it.

At the other end of the spectrum, we had semiformal and informal models such as glossaries, lexical vocabularies, taxonomies, etc. I'll discuss these in more detail below, but the essential point is that these models were *designed for domain experts to standardize the way domains are described* rather than for *execution by computers*.

SKOS: The Glue Between Informal Models and Ontologies

The most important point in Figure 12-1 is the Simple Knowledge Organization System (SKOS) (W3C SKOS Working Group, 2005). SKOS is an OWL vocabulary designed to build informal models in a way that the OWL

reasoner can understand, but without constraining domain experts to think only in terms of logic and set theory. OWL can represent semiformal models such as lexical vocabularies and taxonomies. The benefit of utilizing OWL to do so is that it's still very intuitive to domain users but can also be understood and transformed into ontologies. That is, a semiformal model can be the first step toward an ontology.

SKOS was designed precisely to occupy the space between purely informal artifacts such as glossaries and fully formal ontologies. SKOS provides a lightweight, standards-based way to represent structured domain knowledge that is both intuitive for humans and usable by software, without requiring the full logical commitments of an ontology.

At its core, SKOS is centered on the notion of a concept. A `skos:Concept` is a unit of meaning. Concepts are grouped into concept schemes, which provide contextual boundaries such as a controlled vocabulary for tags, a taxonomy for product categories, or a thesaurus for regulatory terms. This notion of explicit scoping is critical when working in enterprise environments, where the same term may legitimately carry different meanings in different contexts.

SKOS supports a small but powerful set of semantic relationships that enable gradual formalization:

- Hierarchical inverse relations, such as `skos:broader` and `skos:narrower`, allow concepts to be organized into taxonomies (recall that in graph theory, a taxonomy is a directed acyclic graph) without committing to the semantics for the direction of the graph. That is, a `skos:narrower` link between `ConceptA` and `ConceptB` can mean that `ConceptB` is a subclass of `ConceptA`, an instance of `ConceptA`, a part of `ConceptA`, or is contained in `ConceptA`.
- Associative relations, via `skos:related`, capture meaningful connections that are not hierarchical but are nevertheless important for navigation, discovery, and search. These links will typically evolve into object properties and values as the semiformal model is transformed into an ontology.
- Lexical relations, including `skos:prefLabel, skos:altLabel`, and `skos:hiddenLabel`, support synonyms, near-synonyms, abbreviations, and variant spellings in a way that is explicit, multilingual, and machine-readable.

Together, these features make SKOS especially well-suited for modeling lexical vocabularies that define tags (see discussion of Buckets in **Chapter** 7), thesauri, and other semiformal knowledge structures. Unlike free-text tags, SKOS-based tags are governed: they can be curated, versioned, annotated, and aligned across systems. At the same time, they avoid the rigidity and maintenance burdens that often arise when such vocabularies are modeled directly as enumerated ontology classes.

From a methodological perspective, SKOS plays a crucial role in ontology development. Creating an initial SKOS model allows domain experts and knowledge engineers to collaborate on terminology, scope, and conceptual boundaries before introducing formal axioms. Ambiguities, overlaps, and gaps in the domain vocabulary tend to surface early, when they are still inexpensive to resolve. In this sense, SKOS models function as conceptual scaffolding: they stabilize shared meaning while leaving room for evolution.

As ontology development progresses, selected SKOS concepts can be promoted into ontology classes or individuals, relationships can be refined into formally defined properties, and informal hierarchies can be replaced, or supplemented, by logical definitions expressed in OWL. Importantly, this transition does not require discarding the SKOS model. In many mature systems, SKOS and OWL coexist: SKOS supports tagging, navigation, and discovery, while OWL supports reasoning, validation, and integration.

In this way, SKOS acts as the glue between informal, human-centered models and fully formal ontologies. It enables teams to start where domain understanding is strongest (shared language and conceptual structure) while providing a clear and standards-based path toward richer semantic models as requirements mature.

The following provides an overview of semiformal models, as well as a discussion of how SKOS can be used to implement them.

Glossary

A glossary is the simplest kind of semiformal model. It can be as simple as a list of words and definitions for those words. The glossary for this book is available on the book's GitHub wiki: https://github.com/mdebellis/SemanticKG-Design/wiki/Glossary.

In addition, although I won't go into detail about this ontology here, I've also created a SKOS version of the glossary here:

https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_12/skos_glossary.ttl.

I encourage readers to download that file, load it into Protégé, and view it via the Individual **Hierarchy Tab Protégé** plugin and with the Gruff browser or other graph visualization tools.

Controlled Vocabulary (Lexical Vocabulary)

As discussed earlier, vocabulary has a different meaning to Information Architects than to ontologists. The word "vocabulary" as used in this book refers to a reusable ontology. We've been using vocabulary in the Semantic Web sense: a reusable ontology such as SKOS, Dublin Core, or FIBO, designed to model a technical or business domain at a conceptual level. In contrast, *controlled vocabulary* refers to a semiformal list of terms, often curated by domain experts. Sometimes these models are hierarchical, but without the logical structure, reasoning capabilities, or semantic commitments of an OWL ontology. For clarity, I refer to these semiformal vocabularies as *lexical vocabularies* to differentiate them from reusable ontologies.

The shared terminology is not accidental. Many of the widely used Semantic Web ontology vocabularies such as SNOMED for healthcare began their lives as lexical vocabularies: curated lists of standard terms developed for cataloging, metadata, or subject indexing. SKOS itself was created to provide a formal representation for exactly these kinds of knowledge organization systems. The Semantic Web community retained the word "vocabulary" partly out of continuity with this earlier practice, even as the meaning evolved to refer to more rigorous, machine-interpretable ontologies.

A controlled lexical vocabulary modeled in SKOS is a curated list of approved terms used to ensure consistency in naming and categorization. It resolves synonyms, near-duplicates, and ambiguous labels, providing a shared language for both human stakeholders and downstream systems. Controlled lexical vocabularies often begin life as glossaries but go further by defining allowable terms for metadata fields, search facets, forms, and tagging interfaces. In the Ontology Pipeline, the lexical vocabulary is the first step in bringing order to chaotic or inconsistent language.

Metadata Standards

Metadata standards define the structural fields used to describe data and information such as titles, creators, subjects, types, and provenance. These

schemes enforce consistency in how information assets are described, enabling interoperability across systems. Metadata elements can be paired with controlled vocabulary values, bringing stability to the "aboutness" of information. Metadata standards play a crucial role in enterprise classification, content management, and data governance, and they set the stage for the more structured relationships in taxonomies and thesauri.

Taxonomy

A taxonomy organizes concepts into a hierarchical structure using broader–narrower (parent–child) relations. It introduces structure into the controlled vocabulary, creating a navigable hierarchy that supports browsing, tagging, and classification. While often depicted as a tree, real taxonomies commonly allow polyhierarchies (multiple broader/narrower terms), making them directed acyclic graphs (DAGs). Taxonomies are widely used in enterprise search, CMS systems, e-commerce catalogs, design of web sites, and subject classification.

Thesaurus

A thesaurus extends a taxonomy by adding semantic relationships such as synonyms, related terms, associative links, and cross-references. These relationships help users and systems resolve ambiguity, improve search and retrieval, and represent conceptual neighborhoods. SKOS provides a principled vocabulary (`skos:related`, `skos:altLabel`, `skos:exactMatch`) for encoding these richer semantics. As Talisman notes, thesauri represent the beginning of true semantic reasoning in the Ontology Pipeline, bridging classification structures and ontological models.

Folksonomy

A folksonomy is a bottom-up classification generated by user tagging behavior. A familiar example of a folksonomy is the use of hashtags on social media platforms such as Twitter/X, where users independently tag content using their own terms without a shared controlled vocabulary. Unlike controlled vocabularies or taxonomies, folksonomies are emergent: they reflect real patterns of meaning and association in practice rather than curated standards. While messy and inconsistent, folksonomies reveal how users conceptualize topics "in the wild," making them valuable sources of

insight when designing more formal models. They often serve as raw input that later evolves into controlled vocabularies or taxonomies.

Folksonomies have also migrated into the corporate world, where they are often used to gather early input on terminology, concepts, and informal relationships within a domain. In these settings, user-generated tags can provide valuable signals about how practitioners conceptualize a problem space, even when the terminology is inconsistent or ambiguous.

Among practitioners who use folksonomies, there is a long-standing debate about whether they should remain completely unstructured or whether they should be reviewed and refactored to establish conventions and shared vocabularies. As is often the case with such debates, there is no single right answer. For open social media platforms, refactoring a folksonomy is rarely practical and may even be counterproductive, as the value of the system lies precisely in its openness and lack of constraint.

Within an organization, particularly when folksonomies are used as part of a knowledge management or data governance strategy, refactoring can be beneficial. In these cases, a curated or partially normalized folksonomy can serve as a transition point between purely informal tagging and more structured approaches, such as lexical vocabularies. From there, selected portions of the vocabulary may be further refined into formal ontologies, following the ontology pipeline described earlier in this chapter.

Example of an Ontology Pipeline from SKOS to OWL

Semiformal models are a scaffolding that can support ontology design. They remain valuable even after an ontology is complete: users continue to rely on taxonomies for navigation, on glossaries and controlled vocabularies for communication, and on metadata standards to ensure consistent description and retrieval.

Next, we will take an example taxonomy defined in SKOS and show how it can be transformed into a basic ontology. A common example in the Information Architecture space is a corporate organizational taxonomy. Such taxonomies include different logical relations grouped into one tree. Since I'm very familiar with the way consulting firms are structured, I'll use a hypothetical consulting firm for my example taxonomy.

The SKOS hierarchical properties (`skos:broader, skos:narrower`) intentionally represent informal conceptual generalization rather than

formally defined ontological relationships such as `rdfs:subClassOf`, `rdf:type`, or `part_of`.

In practice, enterprise taxonomies frequently mix different kinds of conceptual relationships: classification, specialization, composition, organizational structure, and even instance-like entries, within a single informal hierarchy.

In this example, SKOS is used as the semiformal modeling layer. The initial hierarchy captures intuitive notions of "more general" and "more specific" without prematurely committing to whether a relationship should ultimately be interpreted as subclassing, instantiation, parthood, role restriction, or another formal construct.

The consulting taxonomy example illustrates an important methodological lesson: an informal taxonomy drawn on a whiteboard does not automatically decompose into clean ontological categories. The work of ontology engineering consists precisely in examining each hierarchical link and determining its formal meaning. SKOS provides a useful starting point for this process, but it does not replace formal modeling.

To facilitate viewing the taxonomy in Protégé, I loaded the **Individual Hierarchy Tab** plugin. I encourage readers to also load that Protégé plugin so they can follow along. If you need help loading a plugin, go back to **EXERCISE 1-3: LOADING PROTÉGÉ PLUGINS** in **Chapter 1**.

You can find the initial taxonomy defined with SKOS here: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_12/consulting_skos_taxonomy.ttl.

To load the taxonomy from the URL, remember to use the GitHub *raw* version of the file. See **Loading Ontology Files from GitHub** in **Chapter 5**.

https://raw.githubusercontent.com/mdebellis/SemanticKG-Design/refs/heads/main/chapter_12/consulting_skos_taxonomy.ttl

Load that into Protégé and run the reasoner. Then, use **Window➤Tabs➤Individual Hierarchy** tab option to view the basic taxonomy. Expand the properties in the left side view and select the `skos:narrower` property. Also, select the **Hide Orphans** check box at the top of the right view. Next, expand the individuals in the right view so that

every individual is completely expanded. Your Protégé tab should now look similar to Figure 12-2.

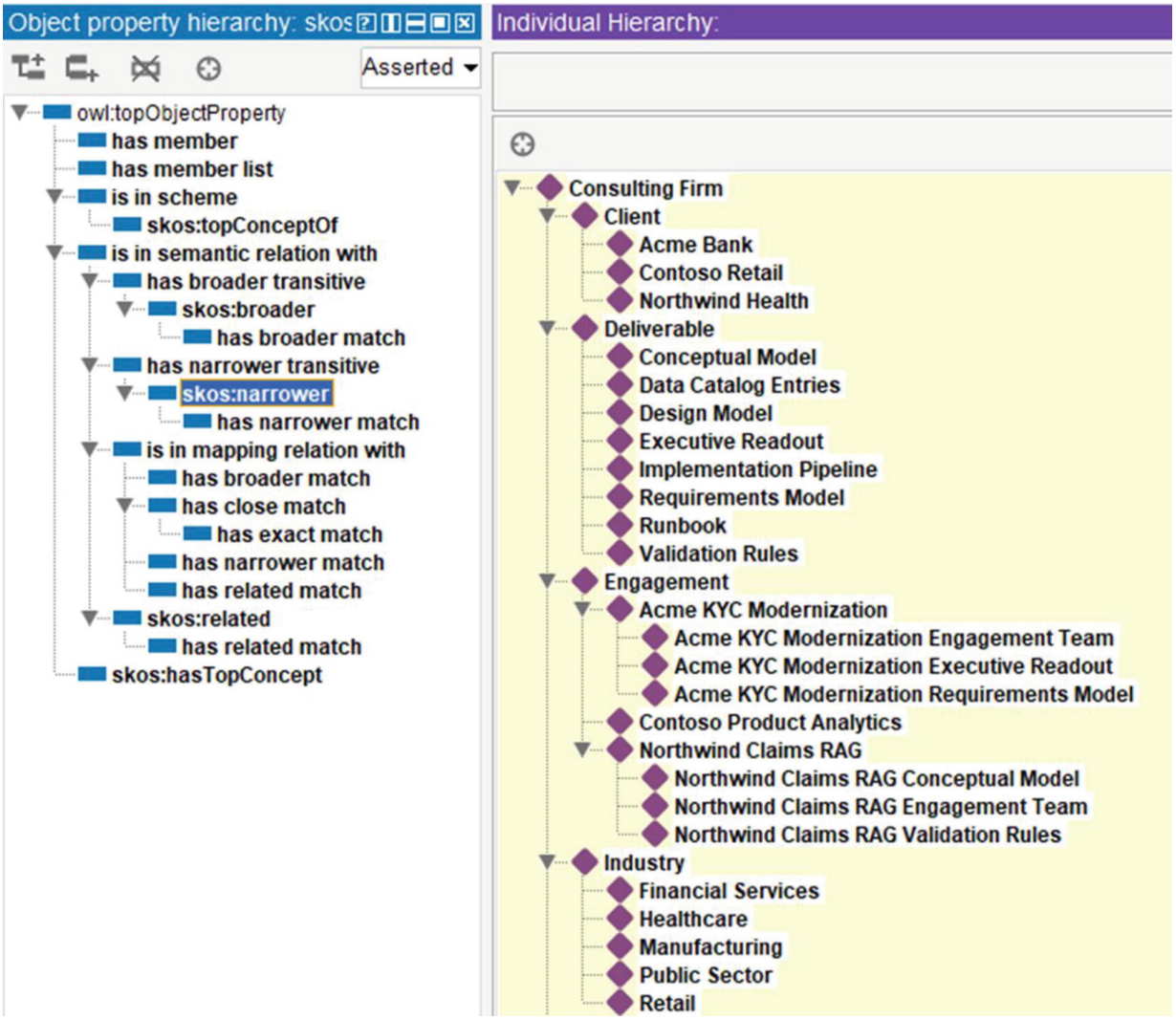


Figure 12-2 The consulting taxonomy in Protégé

The screen print above doesn't show the complete taxonomy, but you should be able to see it in Protégé. Also, you can load it into AllegroGraph. If needed, see the following exercises in the section "**Advanced SPARQL Queries**" in **Chapter 3** to see how to get access to the free version of AllegroGraph and how to load an ontology into AllegroGraph and Gruff:

1. **EXERCISE 3-2: GET ACCESS TO ALLEGROGRAPH**
2. **EXERCISE 3-3: LOADING THE PEOPLE ONTOLOGY INTO ALLEGROGRAPH**

You can also use the book's GitHub Wiki which is a bit easier as it has links: https://github.com/mdebellis/SemanticKG-Design/wiki/Getting-Access-to-the-Community-Version-of-AllegroGraph.

I recommend you execute each of the SPARQL queries and updates in this section. You can find them all in the file: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_12/taxonomy_skos_transformations.sparql.

Due to the limitations of the SPARQL tools in Protégé, the SPARQL transformations need to be performed in AllegroGraph and Gruff or some equivalent tool that implements the full SPARQL specification. In addition, a visual RDF browser such as Gruff that supports tree layout of RDF graphs enables you to visualize the graph a bit better, although the **Individual Hierarchy Tab** plugin in Protégé works pretty well.

Before transforming the taxonomy into an OWL ontology, it is useful to visualize the original SKOS structure exactly as it exists in the source data. The following query retrieves the parent–child relationships defined by `skos:narrower` and restricts them to the concepts that belong to the consulting taxonomy scheme. If you want to follow along in Gruff, you may find these additional Wiki exercises useful: https://github.com/mdebellis/SemanticKG-Design/wiki/TipsForUsingGruffWithAllegroGraph. There are also other useful Gruff and AllegroGraph exercises here: https://github.com/mdebellis/SemanticKG-Design/wiki/Exercises. You can use the SPARQL query shown in Listing 12-1 in Gruff to view the initial taxonomy.

```
PREFIX
:     <https://michaeldebellis.com/consulting_taxonomy/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-
ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX sulo: <https://w3id.org/sulo/>
```

```
# Query to see initial SKOS taxonomy
SELECT DISTINCT ?parent ?child
WHERE {VALUES ?scheme { :Consulting_Firm_Taxonomy }
  ?root  skos:topConceptOf ?scheme .
  ?root  (skos:narrower)* ?parent .
  ?parent skos:narrower ?child .}
ORDER BY ?parent ?child
```

Listing 12-1 A SPARQL query to visualize the original consulting taxonomy

The query begins by selecting the taxonomy scheme and identifying its root concept using `skos:topConceptOf`. From that root, it traverses the taxonomy hierarchy using a transitive path expression, `(skos:narrower)*`, to find all concepts reachable within the scheme.

For each reachable parent concept, the query retrieves its immediate children via `skos:narrower`. The result is a set of (parent, child) pairs that represent the direct edges of the taxonomy tree. These edges can be loaded directly into a visualization tool such as Gruff to display the hierarchical structure without introducing any OWL semantics or inferred relationships.

Because the query explicitly connects the root concept, parent concepts, and child concepts through shared variables, it avoids generating unnecessary intermediate results and remains efficient even as the taxonomy grows.

After running the SPARQL query in Listing 12-1, you will see a table of values returned at the bottom. Select **View➤Graph View in Gruff**. That will show you the results in a graph rather than a table. The result of this query will be messy. To make it look like the diagram below, find the root of the tree which is the `skos:Concept :Consulting_Firm` and select it. Then, use the **Layout➤Do tree layout from selected node** option in Gruff. Your graph should now look similar to Figure 12-3. If your graph goes from left-to-right and you'd prefer it go top-to-bottom, then use the Gruff option: **Visual Graph Options➤Tree Layout Options➤Orient Tree Layouts Vertically.** This will toggle the tree layout so that it uses vertical rather than horizontal layout. Make that change, select the `:Consulting_Firm` node and do **Layout➤Do tree layout from selected node.** Now your graph should go from top to bottom.

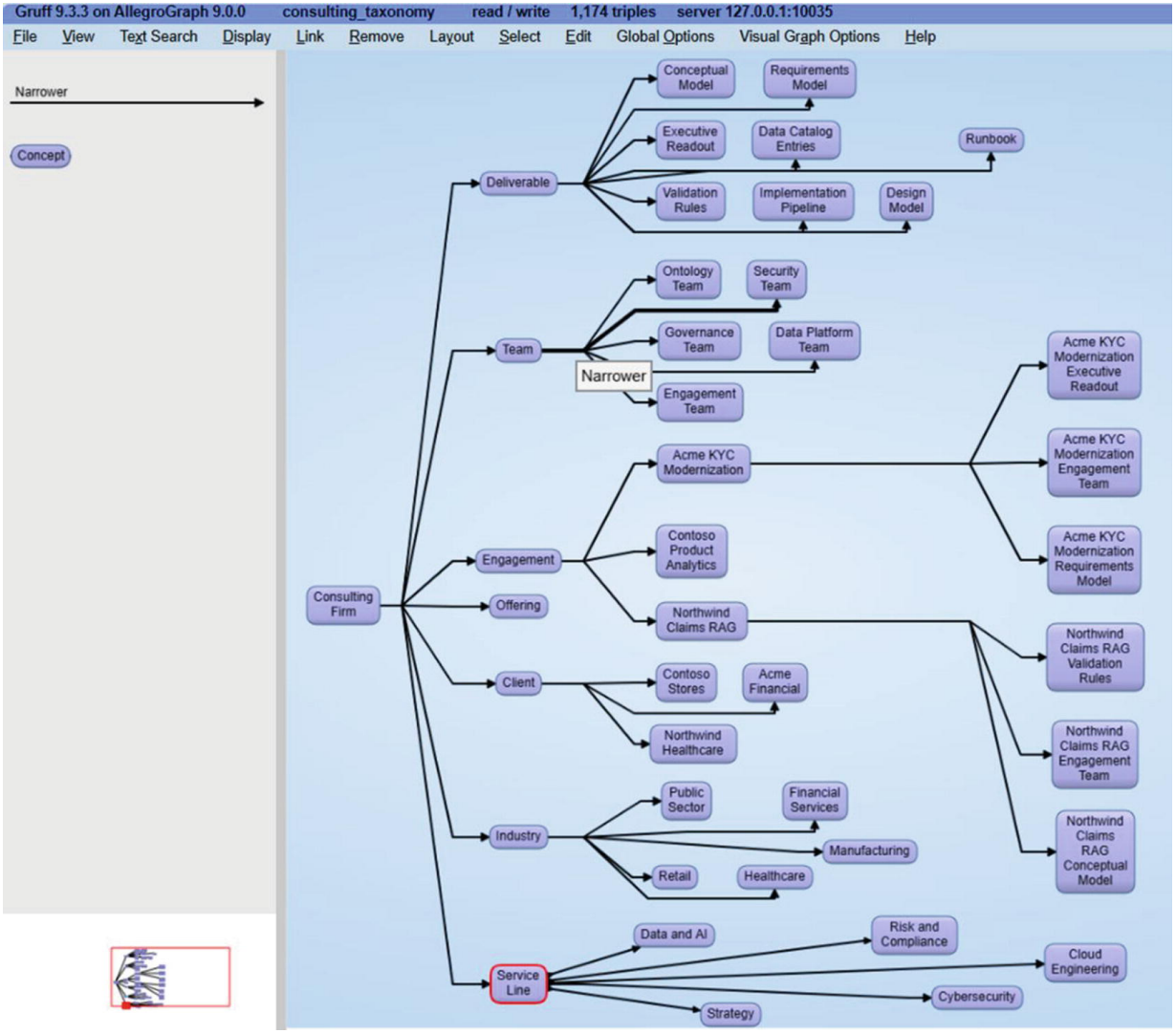


Figure 12-3 The original consulting taxonomy in Gruff

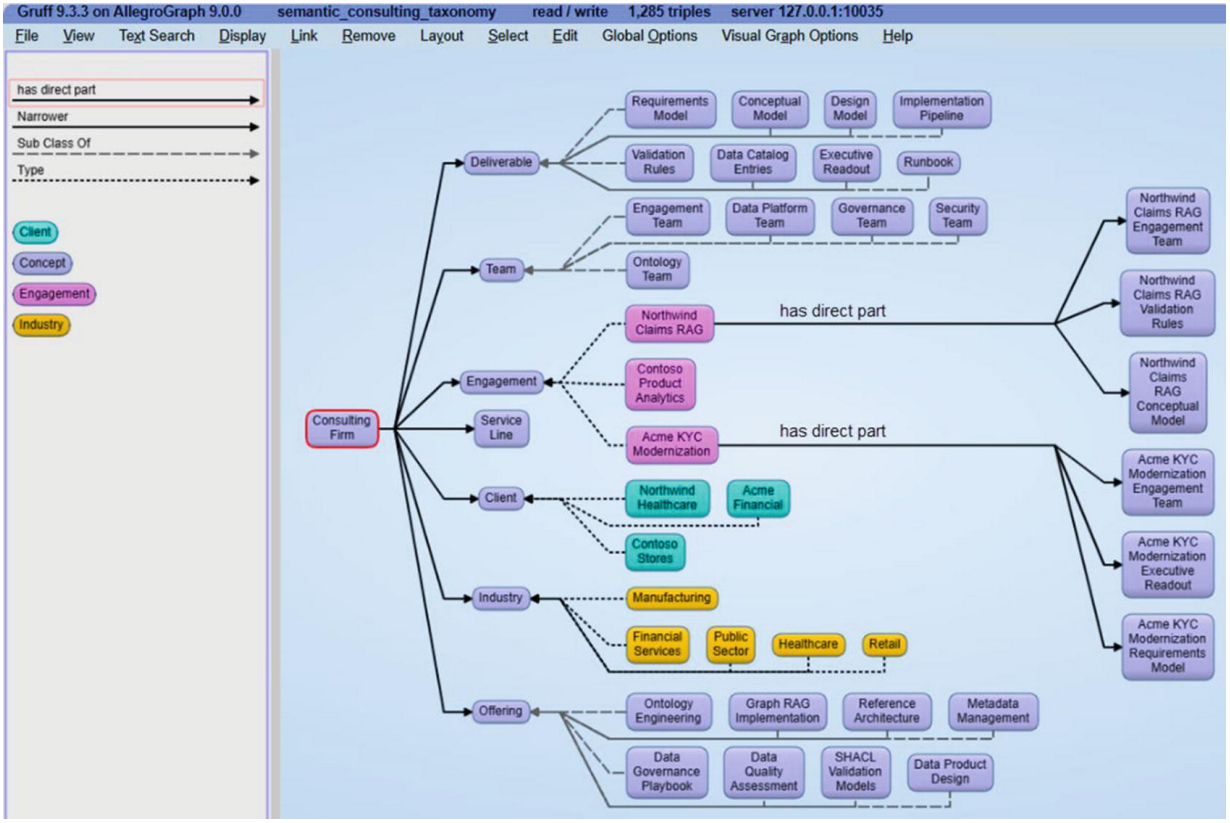


Figure 12-4 The consulting taxonomy with additional semantic properties: type, subclass, and part-of

The query above retrieves a SKOS taxonomy by identifying the root node and traversing the `skos:narrower` property values from that node. Rather than assuming a fixed root node, the query first identifies the root of the taxonomy by retrieving the concept that has been declared as the top concept of the scheme (`skos:topConceptOf`). This is the standard SKOS mechanism for defining where a taxonomy begins. It allows the same query to work even if the root concept changes.

The result is a clean, tree-shaped structure that can be visualized directly in a graph tool such as Gruff. All edges in the resulting graph are expressed uniformly using `skos:narrower`. All nodes are instances of `skos:Concept`. Just as `skos:narrower` is intentionally abstract and can mean subclass, instance, part-of, and other relations, so a `skos:Concept` can be a class, instance, or even a property. The taxonomy is therefore purely organizational and intuitive, providing a human-friendly structure that can later be reinterpreted and refactored into richer OWL semantics without losing the original conceptual organization. Next, we'll see how to transform this taxonomy into an ontology.

Transforming the Taxonomy into an Ontology

The first thing we observe is that :Consulting_Firm and all its immediate children are classes. :Consulting_Firm describes the set of all consulting firms and the various children that are skos:narrower than :Consulting_Firm such as :Client, :Industry, and :Deliverable are also clearly classes.

So, we'll start with a SPARQL INSERT update that turns the root and its direct children into classes. Note that for the following updates you won't see any results because they change the values of the graph but not the visualization. If you look at the bottom of the Gruff Query View, you will see messages such as "Query took N seconds. M triples added." This is an example of how everything in the Semantic Web, including OWL classes and properties, is modeled as RDF graphs. When you define SKOS Concepts to also be OWL classes and subclasses, you are adding more triples to the RDF graph. Select **View➤Query view** so you can enter another query and add some structure to the graph.

Run the update in Listing 12-3 to turn the root class and its immediate children into OWL classes. As always, remember to retain the prefix statements from the first query above.

```
# To begin make the root node of the taxonomy and
# its children into OWL classes
INSERT {:Consulting_Firm rdf:type owl:Class .
                ?concept rdf:type owl:Class .}
WHERE {:Consulting_Firm skos:narrower ?concept .}
```

Listing 12-2 Transforming the root and first level to classes

Note this is an example of punning discussed in **Chapter 3** in the section **"Three Types of Properties."** Punning is necessary to utilize object properties of SKOS (such as skos:narrower and skos:broader), that are defined as inverses and whose semantics are maintained automatically by the OWL reasoner. For example, if :Concept1 skos:narrower :Concept2, then the inverse relationship :Concept2 skos:broader :Concept1 is inferred by the reasoner. SKOS also defines additional object properties, including transitive and hierarchical relations, which we will discuss later in this chapter.

In order to apply these object properties to concepts that will later be treated as OWL classes, we rely on punning. Punning allows a single IRI to be used both as a class and as an individual. From a modeling perspective, this enables us to attach SKOS relationships which are defined as object properties to entities that we also want to treat as classes when we transition to a more formal ontology.

If this seems a bit confusing to you, you aren't alone. It's one of the less intuitive aspects of OWL, and it's why for the most part, I recommend people not use punning.

The reason for exceptions like this case is that SKOS object properties are designed to work with classes and properties as well as with instances. SKOS uses the inferencing of the reasoner applied to object properties. Thus, we cannot take full advantage of SKOS unless we use puns to get those classes and properties to also be individuals.

While punning can appear unusual at first, it is a standard feature of OWL 2. It provides a pragmatic way to combine lightweight semantic structures such as taxonomies with more formal logical models. For readers who are familiar with object-oriented design, think of puns as a way to implement basic meta-class and meta-property capabilities. For readers interested in the details, punning is discussed in the OWL 2 specification (W3C, 2012), and I have written about it in more detail on my blog (DeBellis, 2023a).

In this chapter, the important point is not the logical theory behind punning, but its practical effect: it allows us to use SKOS to capture and maintain hierarchical relationships while deferring more precise ontological commitments (e.g., is a concept a class or an instance) until we are ready.

Now, let's look at the children of `:Client, :Industry` and `:Engagement`. These are clearly instances of their parents. `:Acme_Bank` is an instance of `:Client`, `:Financial_Services` is an instance of `:Industry`, and `:Northwind_Claims_RAG` and its siblings are instances of `Engagement`. Thus, we can use the following transformations in Listing 12-3 (as always, make sure to leave the prefix statements from the first transformation). Also, remember you can't paste all three of these queries. It is possible to create one query that does all three operations at once using UNION, but to keep it simple, I just made three separate queries. But that means you need to do each one separately. As with the previous query, all you will see are messages such as "The triple count increased by 10" at the bottom of the screen.

```
# The children (skos:narrower) of Client are instances.
INSERT {?client rdf:type :Client.}
WHERE {:Client skos:narrower ?client.}

# The children (skos:narrower) of Industry are
instances.
INSERT {?industry rdf:type :Industry.}
WHERE {:Industry skos:narrower ?industry.}

# The children (skos:narrower) of Engagement are
instances.
INSERT {?engagement rdf:type :Engagement.}
WHERE {:Engagement  skos:narrower ?engagement.}
```

Listing 12-3 SPARQL to transform the appropriate narrower links to rdf:type

Next, let's look at the relation between `:Deliverable, :Offering`, `:Service_Line`, and `:Team` and the relation each of them has with their `skos:narrower` children. These are subclass relations. A `:Conceptual_Model` is a kind of `:Deliverable`; `:Data_Product_Design` is a kind of `:Offering`, and so on. As a result, our next transformation is shown in Listing 12-4.

```
# Rest of the children of the second level concepts are
subclasses
INSERT {?child rdfs:subClassOf ?parent .}
WHERE {SELECT DISTINCT ?child ?parent
      WHERE {
      {:Deliverable skos:narrower ?child .
BIND(:Deliverable AS ?parent)}
      UNION
      {:Offering skos:narrower ?child . BIND(:Offering
AS ?parent) }
      UNION
      {:Service_Line  skos:narrower ?child .
BIND(:Service_Line AS ?parent)}
      UNION
```

```
      {:Team  skos:narrower ?child . BIND(:Team AS ?
parent)}}}
```

Listing 12-4 Transforming Appropriate Narrower Links to Subclass Relations

This update uses a single `INSERT` statement together with a structured `WHERE` clause to transform the appropriate `skos:narrower` edges into OWL subclass relationships.

At the outer level, the `WHERE` clause contains a `SELECT DISTINCT` subquery that produces a stream of (child, parent) pairs. Each pair represents a single subclass assertion that should be added to the ontology. Using a subquery (a `SELECT` inside the outer `WHERE` clause) makes the intent explicit: first compute which relationships should exist, then insert them.

Inside the subquery, the `WHERE` clause is composed of several alternative patterns combined using `UNION`. Each branch of the `UNION` matches the children of one second-level taxonomy concept (e.g., `:Deliverable` or `:Service_Line`). The `BIND` expression in each branch associates the matched child with its corresponding parent class. As a result, every solution row has exactly one child and exactly one parent.

The only children that are beneath the second level left are the children that are `skos:narrower` than the engagement instances. For example, `:Northwind_Claims_RAG` has children `:Northwind_Claims_RAG_Conceptual_Model`, `:Northwind_Claims_RAG_Engagement_Team`, and `:Northwind_Claims_RAG_Validation_Rules`.

These aren't instances of the engagement. A conceptual model is not an instance of an engagement. They also aren't subclasses. A conceptual model is not a kind of engagement. The correct choice is that each of these children is part of its parent. An engagement team is part of an engagement. A model developed in an engagement is part of the engagement. Thus, we'll use the `sulo:hasDirectPart` object property.

To do that, we first have to add the property to the ontology (Listing 12-5).

```
# Add object property before using it
INSERT DATA {sulo:hasDirectPart rdf:type
owl:ObjectProperty ;
                      rdfs:label "has direct part"@en .}
```

Listing 12-5 Adding a has Part Object Property to the Ontology

Then, we can do the final transformation to add the part-whole relations (Listing 12-6).

```
# The children of Engagements are parts of the
engagement
INSERT {?engagement sulo:hasDirectPart ?part .}
WHERE {{ :Acme_KYC_Modernization  skos:narrower ?part .
BIND(:Acme_KYC_Modernization  AS ?engagement)}
  UNION
  {:Northwind_Claims_RAG skos:narrower ?part .
BIND(:Northwind_Claims_RAG AS ?engagement)}}
```

Listing 12-6 Using hasDirectPart to Transform Narrower Links to Subpart Relations

To see the whole graph with color coding, I encourage readers to check out the extra exercise on the book's GitHub site: https://github.com/mdebellis/SemanticKG-Design/wiki/ConsultingTaxonomyToOntology. Following the instructions on that exercise, your graph should now look like Figure 12-4.

Also, if you haven't been using the SPARQL transformations, you can load what the state of the graph should now be from the file https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_12/consulting_semantic_taxonomy.ttl and do the same manipulations in that optional wiki exercise file.

The Root-Level Problem

Up to this point, the transformations from the SKOS taxonomy to OWL have been fairly direct. For most branches of the taxonomy, each `skos:narrower` relation could be refactored into one of three familiar OWL patterns:

- A "kind of" subclass (`rdfs:subClassOf`) relation
- An "is a" instance-of (`rdf:type`) relation
- A "part of" part–whole (`sulo:hasDirectPart`) relation

In each case, the resulting OWL graph both preserved the shape of the taxonomy and added more precise semantics. However, the relationship

between the root concept `:Consulting_Firm` and its immediate children turns out to be fundamentally different.

The concepts directly below `:Consulting_Firm` (`:Client`, `:Engagement`, `:Deliverable`, `:Industry`, `:Offering`, `:Service_Line`, `:Team`) are not subclasses of a consulting firm, not instances of it, and not parts of it in any simple mereological sense.[1] These relations from the root concept don't cleanly map into any of the three most common ways to map `skos:narrower` to OWL. Instead, they represent roles, categories, and organizing dimensions that characterize what a consulting firm does and how it operates.

This is the first place where a clean, edge-for-edge refactoring from SKOS into OWL breaks down. This is not a failure of the example. It is precisely the point.

At the SKOS level, the use of `skos:narrower` from Consulting Firm to these second-level concepts is entirely appropriate:

- It produces a clear, navigable conceptual hierarchy.
- It matches how people naturally think about the domain.
- It supports visualization, browsing, and early-stage modeling.
- It avoids prematurely committing to formal semantics.

For purposes such as diagrams, exploration in Gruff and other graphing tools, and communication with domain experts, retaining these `skos:narrower` links is not only acceptable but ***desirable***.

In other words, SKOS is doing exactly the job it was designed to do: function as a tool for Information Architects who want to create structures that are easy for nontechnical people to understand. From an OWL perspective, however, the relationship between `:Consulting_Firm` and its immediate children is not a binary relation between individuals, parts, or classes. It is a set of class-level axioms describing what must exist for something to count as a consulting firm.

For example (using informal language):

1. A consulting firm has clients.
2. A consulting firm runs engagements.
3. A consulting firm serves one or more industries.

4. An engagement has a client.
5. An engagement produces deliverables.
6. Clients have an Industry.
7. A consulting firm offers various services.

In OWL, statements like these are modeled as axioms, typically using existential restrictions. Conceptually, they are as shown in Listing 12-7.

```
1.  Consulting_Firm ⊑ has_client some Client
2.  Consulting_Firm ⊑ runs_engagement some Engagement
3.  Engagement ⊑ has_client some Client
4.  Engagement ⊑ produces_deliverable some Deliverable
5.  Client ⊑ has_industry some Industry
6.  Consulting_Firm ⊑ offers_service some Service_Line
```

Listing 12-7 Some Narrower Links Don't Directly Transform to a Simple OWL Relation

The ⊑ character is another way to define a subclass. This is what you do when you add axioms to a class. You define anonymous classes that are superclasses of the class with the axioms. The important point is these are axioms on the classes of the emerging ontology, not edges in a graph. OWL class axioms are represented internally as RDF, but not as direct edges between classes. Rather they are modeled as blank nodes with structures that model logical expressions. See "**OWL: A Peek Under the Hood**" in **Chapter 3** for some examples.

The result is a pragmatic and intentional split:

- SKOS is used to represent the conceptual organization of the domain, including the root-level structure.
- OWL is used to add formal semantics.
- At the root level, SKOS remains the primary representation for structure and visualization.
- OWL axioms are introduced to capture the deeper meanings and semantics of the domain, even though those axioms do not appear as simple edges in a graph.

This is not a workaround. It is an example of layered modeling, and it illustrates exactly why semiformal models are so valuable as a first step.

The Bigger Lesson

The key takeaway is that different modeling questions require different representational tools. For Information Architects, flexible relations such as `skos:narrower` are usually the appropriate model. To transform an intuitive taxonomy into an ontology with more rigorous semantics, we need to analyze the SKOS relations and determine which specific OWL relation is best. In addition, there will be examples where there is no simple OWL modeling relation that a SKOS term maps to. Instead, we will also need to transform them into axioms.

Even in this small example, we encounter relationships that

- Make perfect sense conceptually
- Are easy to express informally
- Do not map cleanly to graph edges in OWL

When constructing this example, my initial assumption was that the entire SKOS taxonomy could be systematically refactored into a small set of OWL relations: subclass, instance-of, and part–whole. In this case, however, that assumption breaks precisely at the most intuitive point: the root of the taxonomy. At first glance, this felt like a flaw in the example. However, as I thought a bit more carefully, I realized that the fact that the initial taxonomy didn't completely map to ontology relations made this example ***illustrate the difference between modeling as an Information Architect vs. modeling as an ontologist***.

SKOS allows us to model those intuitive relationships, while OWL allows us to add precision where it belongs, without forcing everything into a single tree. That tension is not a problem to be solved. It is the reason the ontology pipeline exists.

What the example demonstrates is that intuitively meaningful conceptual structure does not always correspond to single, well-formed logical relations. The concepts immediately below Consulting Firm are not subclasses, not instances, and not parts; they are dimensions along which consulting firms are organized and described and need to be modeled as axioms on the classes in the ontology.

The final step is to model the axioms that are based on the remaining `skos:narrower` relations into OWL axioms on classes such as `Consulting_Firm`, `Client`, and `Engagement`. This takes us from our current taxonomy to a model where all the initial informal relations are now modeled in OWL. That is, we now have a (very basic) Consulting Ontology. I'm not going to walk through each step. However, you can load the following file into Protégé to see the final result: https://github.com/mdebellis/SemanticKG-Design/blob/main/chapter_12/consulting_ontology.ttl.

You will see that I've added several new properties, axioms, and an instance of a hypothetical consulting firm called `Brachman_Consulting`. Some of these axioms and property values on an example instance called `Brachman_Consulting` are shown in Protégé in Figure 12-5.

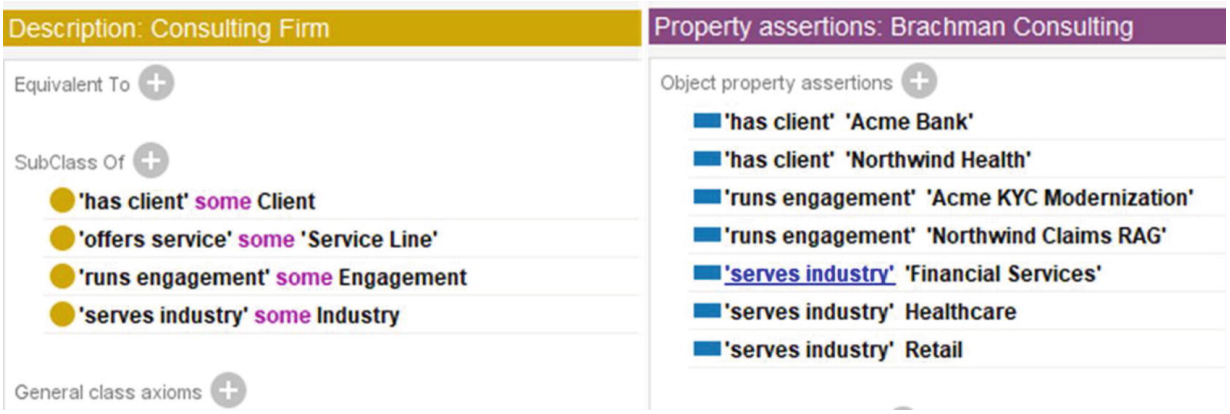


Figure 12-5 Axioms defined on the Consulting_Firm class and Example Values on an Example Instance

I encourage the reader to also load that final OWL file into Protégé and to view the classes that now have axioms and the various instance values.

Other SKOS Object Properties

So far, we've deliberately focused on `skos:narrower` in order to keep the example simple and concrete and to show how a semiformal SKOS model can be refactored into an ontology. Having shown how that works in practice, it is worth briefly stepping back to look at the broader set of object properties defined by SKOS as well as the kinds of problems they are designed to address.

SKOS is a vocabulary specifically designed to support semiformal models and mappings between conceptual schemes, especially in environments where multiple models coexist. To see all of these SKOS properties, load the

consulting taxonomy into Protégé, go to the **Object Property Tab** and expand all of the SKOS object properties. You should see the properties in Figure 12-6.

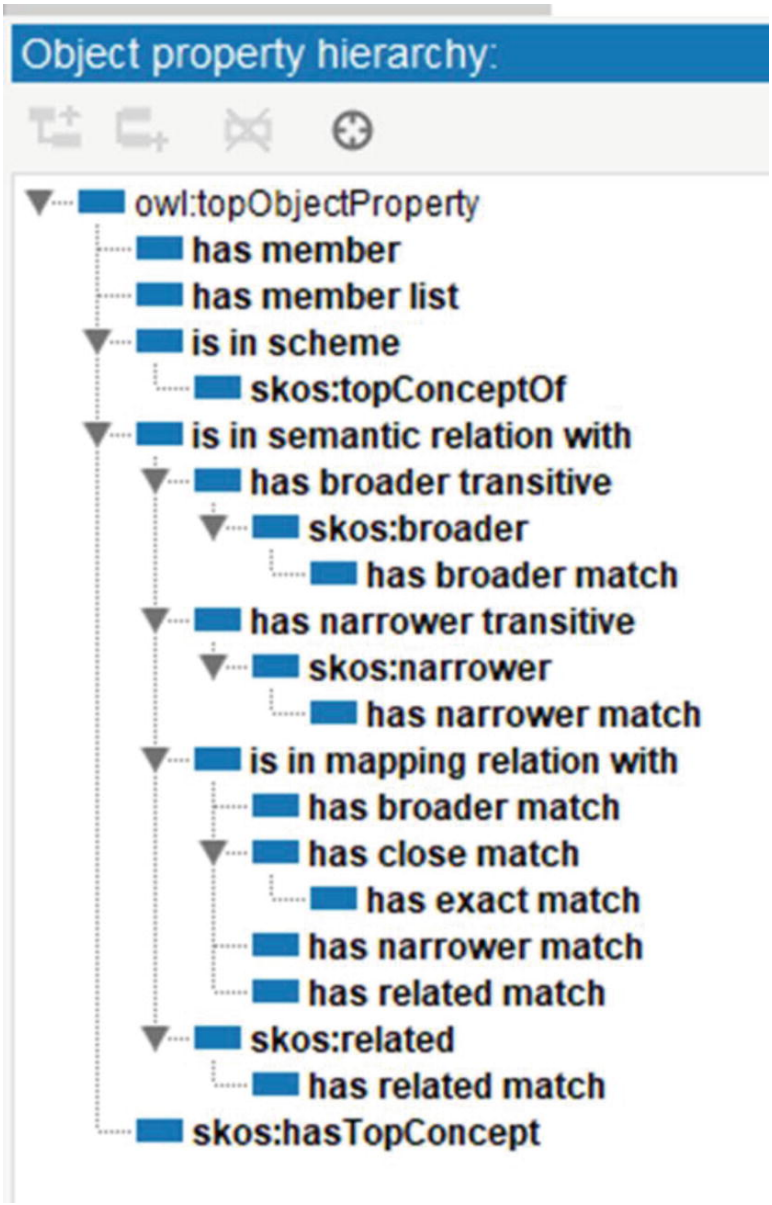


Figure 12-6 SKOS object properties

Hierarchical Relations

The most commonly used SKOS relations are the hierarchical ones:

- `skos:narrower` and `skos:broader`
- `skos:narrowerTransitive` and `skos:broaderTransitive`

The non-transitive properties (`skos:narrower, skos:broader`) are intended to capture direct hierarchical links, while the transitive variants allow tools and reasoners to compute the full closure of the hierarchy. Recall we've seen this pattern before in the ontology patterns described in **Chapter 7**.

Associative Relations

SKOS also supports nonhierarchical associations through `skos:related`.

This property is used to indicate a meaningful conceptual relationship that is not hierarchical. For example, in a healthcare vocabulary, a concept such as `Diabetes_Mellitus` might be `skos:related` to `Insulin_Therapy`. These links are often valuable for navigation, discovery, and user interfaces even when they do not correspond to a well-defined logical property.

Mapping Relations

Some of the most frequently used SKOS properties are designed explicitly for mapping concepts across different vocabularies or models:

- `skos:exactMatch`
- `skos:closeMatch`
- `skos:broadMatch`
- `skos:narrowMatch`
- `skos:relatedMatch`

These properties are especially important in domains such as healthcare, finance, and government that utilize formal lexical coding vocabularies. It is often the case that there will be multiple coding systems and terminologies. For example, healthcare organizations routinely need to relate concepts across ICD, SNOMED CT, LOINC, CPT, and internal clinical vocabularies. In many cases, there is no single correct logical equivalence between concepts. A code in one system may correspond approximately, but not exactly, to a concept in another.

SKOS mapping relations allow these correspondences to be captured explicitly. A `skos:closeMatch` acknowledges practical equivalence for many use cases without asserting logical identity. A `skos:broadMatch` or `skos:narrowMatch` records known differences in scope without forcing a formal subsumption axiom. This makes SKOS an ideal tool for interoperability

scenarios where alignment is incremental, contextual, and often negotiated rather than absolute.

Concept Schemes and Membership

Finally, SKOS provides properties such as

- `skos:inScheme`
- `skos:hasTopConcept`
- `skos:topConceptOf`

These properties allow concepts to be grouped into concept schemes and support navigation and visualization tools such as the Protégé hierarchy shown in Figure 12-2. Concept schemes make it possible to manage multiple overlapping vocabularies, track provenance, and evolve models over time without requiring a single monolithic ontology.

Taken together, these SKOS object properties illustrate why SKOS plays a critical role in semantic architectures. It is not merely a stepping stone to OWL, but a complementary modeling layer that supports classification, alignment, and gradual formalization in complex, real-world domains.

Summary

- *Semiformal models are designed to define domain terminology and present that terminology in a way that is intuitive to users.*
- *Semiformal models include Taxonomies, Glossaries, Thesauri, and Folksonomies.*
- *The Simple Knowledge Organization System (SKOS) is a W3C reusable ontology designed to model semiformal models*. It is one of the most common ontologies in use today.
- *Semiformal models do not compete with ontologies*. Each is useful and each solves different problems.
- *Modeling semiformal models using SKOS is an excellent way to begin by creating something useful and intuitive* and also provide the foundation for an ontology with more semantic representation.
- *In this chapter, I began with a taxonomy of a consulting firm that I created where I purposely did not think about the semantics of the taxonomy links*. I modeled the taxonomy in SKOS and showed how it could be transformed into an OWL ontology using SPARQL and some manual changes. This

chapter is an attempt to implement a basic example of the ideas of Jessica Talisman in her article *The Ontology Pipeline*, and I want to acknowledge and thank her for that article, which helped me understand semiformal models in a way I never could in the past.

- *Further Reading*: *The Ontology Pipeline* by Jessica Talisman (Talisman, 2025).

Footnotes

1 Mereology is the analysis of part-whole relations.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_13

13. Data Products and Domain-Driven Design: A Graph RAG Data Catalog

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

Data Products and Domain-Driven Design: A Graph RAG Data Catalog

> *The greatest value of a domain model is that it provides a ubiquitous language that ties domain experts and technologists together.*
> —Martin Fowler: Foreword to Domain-Driven Design: Tackling Complexity in the Heart of Software (Evans, 2003)

Modern enterprises increasingly rely on distributed architectures for data systems and governance. This shift reflects the same forces that led to the adoption of microservices: the need to scale systems, enable rapid change to take advantage of new business opportunities, align ownership with expertise, and reduce bottlenecks caused by centralized control.

As introduced in **Chapter 11**, a data product is a domain-owned, self-contained unit that exposes data through well-defined, versioned interfaces called ports. It is designed for consumption by other systems and users. Data products are the data-centric counterpart to microservices. In addition, data products bridge the gap between operational and analytic systems.

This chapter builds on that foundation by examining how data products are structured and governed within domains. In particular, it explores how Domain-Driven Design (DDD) provides the framework needed to design data products that are coherent, scalable, and aligned with business meaning. In addition, we'll look at how combining retrieval-augmented generation (RAG) architectures with Semantic Knowledge Graphs can provide truly revolutionary capabilities for organizing and searching enterprise content.

I'll use the term "search," but the Graph RAG architecture goes far beyond traditional search and can perform reasoning, planning, and problem-solving.

Our example is a data catalog, but the same ideas apply to data lakehouse, data mesh, data fabric, and knowledge management systems.

All of the diagrams in this chapter are from the prototype data catalog that can be found here:

https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/knowledge_graph/streamforge_data_catalog_V1.ttl

Note that the knowledge graph and supporting tools related to the data product catalog have their own directory on the book's GitHub site: https://github.com/mdebellis/SemanticKG-Design/tree/main/data_product.

This differs from the convention used elsewhere in this book of placing relevant artifacts in a directory that corresponds to the chapter number. We do so because there is more code and a more complex knowledge graph that are used in **Chapters 11 and 13**. Also, the data catalog example is meant to be more than just an example of some concepts in the book. It is meant to be an example that can be useful as a foundation for future work.

A central idea in DDD is that the domain model provides the foundation for a shared vocabulary, referred to as the *ubiquitous language*. This language should be more than a documentation artifact; it is the means by which domain experts and developers communicate

about concepts, rules, and relationships. The terms used in conversation, design discussions, and code are expected to align, so that the meaning of the system is consistent across all forms of expression. The concept of a ubiquitous language was first described in the USAF-sponsored Knowledge-Based Software Assistant (KBSA) research program, where it was described as a Wide-Spectrum Language (Green et al., 1983). The idea of both is that a collection of integrated languages can describe software assets from the domain level to design and code. In addition, the ubiquitous/wide spectrum language captures the intent of developers and users by documenting the process as well as the software assets. For example, the language captures design changes, the reasons for them, the artifacts that were affected, etc.

Semantic technologies provide a natural way to make this idea explicit for data. OWL defines the concepts and relationships that make up a conceptual domain model and provides reasoners that validate and add knowledge to the model. SHACL models the data integrity constraints and provides a reasoner to find violations of those constraints. SPARQL and LLMs provide the tools to search and reason over the models and data. As we've seen in some simple examples, SPARQL also provides a powerful transformation engine to transform the models and data. An SKG-powered RAG architecture allows important parts of the DDD ubiquitous language to be explicitly modeled as a machine-readable model that can be validated, queried, reasoned over, transformed, and shared across systems.

An important caveat: although our focus is on semantic technologies, they are not the entire ubiquitous language. In DDD, the ubiquitous language is expressed across many artifacts and practices: source code, API contracts, user stories, UML diagrams, Entity/Relationship models, Interface Definition Languages, OpenAPI specifications, GraphQL schemas, and other design artifacts are still essential. These models often capture implementation structure, interaction patterns, service boundaries, or interface details that are not the primary purpose of an ontology.

The role of the semantic model is therefore not to ***replace*** these artifacts, but to provide a ***consistent semantic foundation*** for them. Semantic models define the core terms, relationships, constraints, identifiers, and governance rules that other design artifacts refer to and align with.

At the core of Domain-Driven Design is the concept of the *bounded context*: a well-defined boundary within which a particular domain model applies. From the perspective of data architecture, a data product can be understood as the realization of a bounded context's data responsibilities. It encapsulates not only the data itself, but also the contracts, semantics, and governance rules that define how that data is produced and consumed.

This alignment has important implications. By mapping data products to bounded contexts:

- Ownership is clearly assigned to domain teams.
- Interfaces reflect domain-specific meaning rather than generic schemas.
- Changes can be made to a data product with minimal disruption to the system.

In Semantic Knowledge Graph (SKG) architectures, this alignment extends further. The ontology associated with a bounded context becomes part of the data product itself, *ensuring that semantics—i.e., a clear, formal declaration of meaning for both human and machine interpretation—are an integral part of the ubiquitous language*.

This chapter explores the relationship between Domain-Driven Design and data products utilizing the StreamForge Data Catalog for examples. Bounded contexts define the boundaries of domains. Data products provide the mechanism by which that data is shared across boundaries. Semantic models and knowledge graphs play a critical role in making this relationship explicit, in a form that is both human-understandable and machine-readable.

The goal of this chapter is not to provide a comprehensive treatment of DDD, nor to advocate for any single architectural strategy

such as Data Mesh. Instead, this chapter seeks to provide the minimal set of concepts needed to understand

- How data products, bounded contexts, and semantic models fit together
- Why semantic technology is so strongly aligned to the modeling requirements of DDD

This chapter will show how OWL, SWRL, and SHACL provide Very High Level Languages (VHLLs) that can be the foundation for the *ubiquitous language* in Domain-Driven Design.

Bounded Contexts, Ubiquitous Language, and Explicit Models

We begin with bounded contexts, which define the domain boundaries of data products. At the core of Domain-Driven Design (DDD) is the recognition that large systems fail less often because individual components are poorly implemented than because the *meaning* of concepts becomes ambiguous as systems grow. Terms that appear identical at a superficial level, such as *customer*, *user*, *order*, *account*, and *video*, often refer to different things in different parts of an organization.

For decades, enterprise data management has searched for the silver bullet (Brooks, 1987) of a single integrated enterprise data model. This goes back at least to mainframe-era data dictionaries, COBOL copybooks, corporate data models, and later metadata repositories. The problem is not that the goal of a unified enterprise data model is difficult to realize. The deeper problem is that one unified enterprise data model is the wrong model.

Terms such as *Customer, Product, Order*, or *Account* may look universal, but in practice, they are often *polysemes*: the same word used across domains with overlapping but different meanings (Dehghani, 2022). A single enterprise model tries to eliminate that variation. This often leads to long pointless debates that suck up time but contribute

little business value. A more realistic semantic architecture makes the variation explicit and then provides governed ways to map, compare, and integrate across domains. That's the essence of DDD.

A bounded context defines the scope within which a particular domain model applies, including the concepts it contains, the relationships among them, and the assumptions they embody. Within a bounded context, terminology is precise and internally consistent. Across bounded contexts, differences in meaning are *expected* and *explicitly acknowledged* rather than treated as defects to be eliminated. We'll show some examples of this in the following sections.

Bounded contexts are an essential part of the *ubiquitous language*. From an architectural perspective, bounded contexts play both *organizational* and *technical* roles. Organizationally, a bounded context aligns with a team or business unit, establishing clear ownership and accountability for both implementation and meaning. Technically, a bounded context encapsulates its internal models and assumptions, exposing only selected aspects of its capabilities through explicit interfaces. This dual role is what allows systems to evolve locally without requiring global coordination at every change.

From the perspective of semantic modeling, bounded contexts are *explicit constraints on meaning*. Each bounded context defines a local conceptual space in which terms have specific interpretations and invariants can be enforced. Without semantic technology, these conceptual boundaries are often implicit, encoded informally in design documents, naming conventions, or developer intuition. As systems scale, this implicitness becomes a liability.

Semantic technologies offer a way to make these boundaries explicit and inspectable. Ontologies provide a precise mechanism for defining the concepts and relationships that make up the ubiquitous language. Knowledge graphs provide a flexible substrate for representing instances of those concepts and connecting them across contexts without collapsing them into a single global schema. Rather than enforcing uniformity, *semantic models allow multiple domain models to coexist while still supporting controlled integration*.

This perspective reframes DDD modeling as a form of *conceptual* data modeling as defined in **Chapter** 7. Instead of focusing solely on classes, services, or APIs, it emphasizes the explicit representation of meaning, ownership, and boundaries. When combined with semantic models, DDD concepts become inspectable formal artifacts rather than informal diagrams or prose descriptions.

A key reason DDD aligns so naturally with semantic approaches is that bounded contexts are concerned with modeling *domains*, *not software components*. The primary artifacts of DDD are not classes, services, or APIs, but concepts, relationships, and business logic. These artifacts exist at a level of abstraction above any particular implementation, and often persist even as technologies, platforms, and deployment strategies change.

Recall that in this book, the term *semantic technology* is used broadly to include not only knowledge representation languages such as RDF and OWL but also constraint and validation mechanisms such as SHACL, rules defined with languages such as SWRL, boundaries defined by named subgraphs, and query/transformation languages such as SPARQL. Together, these technologies support the full life cycle of domain modeling: defining concepts, validating their use, integrating across boundaries, and evolving models over time. At the implementation level, these semantic models are often exposed through standard interface technologies such as GraphQL and REST APIs, extending the ubiquitous language of the domain into operational systems.

This combination allows bounded contexts to be treated as first-class semantic artifacts rather than informal design constructs. Domain concepts can be defined precisely, constraints can be rigorously defined and enforced, and relationships across contexts can be modeled explicitly rather than embedded implicitly. As a result, architectural intent becomes explicit, inspectable, queryable, and evolvable.

Those virtues become essential when data products, rather than code alone, become primary integration artifacts. Within a bounded context, data is produced and interpreted according to that context's

ubiquitous language. When data must cross context boundaries, it should not do so implicitly or by exposing internal storage structures. Instead, bounded contexts expose selected data through *data products and ports*.

A data product must be owned by exactly one bounded context. It exposes data through one or more *ports*, each of which defines an explicit interface for access. These interfaces specify how data may be discovered, queried, input, or consumed, independent of how it is implemented internally. The relationship between bounded contexts is governed by explicit *contracts* that define obligations, guarantees, and expectations between producers and consumers. To object-oriented software engineers, this should all seem very familiar. It is the well-understood object-oriented concept of encapsulation.

By making bounded contexts, data products, ports, interfaces, and contracts explicit, organizations can localize meaning while still enabling integration. The boundaries that protect teams from unintended coupling become first-class architectural elements.

Once bounded contexts are defined, the remaining challenge is understanding how those contexts relate to one another. That relationship is captured by *context maps*, which describe not just that contexts interact, but how responsibility for meaning and evolution is distributed across those interactions.

Context Maps and Cross-Context Integration

Bounded contexts localize meaning. That localization is what makes systems and organizations scalable: teams can evolve their models independently, reason clearly about their own domains, and make changes without coordinating globally. However, no bounded context exists in isolation. They must exchange information and integrate capabilities across domains.

DDD addresses this through the concept of the *context map*. A context map describes the relationships between bounded contexts, making explicit how models interact, where translation is required, and which context takes responsibility for defining meaning. Rather than

treating integration as an afterthought, context maps acknowledge it as a first-class design concern.

In practice, bounded contexts interact in a variety of ways. One context may act as an upstream producer of information that another consumes downstream. A shared language may be published for others to adopt. In some cases, translation layers are required to protect one context from assumptions embedded in another. DDD concepts modeled with semantic technology capture these patterns as *explicit architectural relationships defined with OWL, SHACL, and Subgraphs* that shape how systems evolve over time. For example, `Context_Map_Relationship` is an example of the n-ary relation pattern.

Context maps model semantic boundaries and dependencies. Each bounded context defines its own conceptual vocabulary. *Context map relationships* describe how those vocabularies relate: where meanings align, where they diverge, and where explicit mediation is required. Making these relationships explicit helps prevent a common failure mode in large systems, where implicit assumptions about shared meaning gradually accumulate technical debt until integration becomes fragile and opaque.

Semantic knowledge graphs provide a natural way to model context maps as *inspectable artifacts*. Bounded contexts are modeled as explicit entities. Relationships between them are represented as first-class entities. These relationships can carry additional structure, including relationship types, obligations, and contracts that govern how data and meaning cross context boundaries. This allows architectural assumptions about integration to be queried, validated, automated, and evolved alongside the systems they describe.

Importantly, context maps do not prescribe a single integration strategy. They do not require centralized schemas, nor do they imply fully decentralized architectures. Instead, they provide a shared conceptual framework for reasoning about integration choices. Different relationships may call for different approaches, and those approaches may change over time as organizations evolve.

This perspective becomes especially important when data is involved. Data often crosses bounded context boundaries more frequently and more persistently than behavior. Reports, analytics, vector embedding models, databases, and downstream applications routinely depend on data produced by multiple domains. Without explicit modeling of cross-context relationships, these dependencies tend to accumulate invisibly, reintroducing global coupling through shared data structures.

Context maps make these dependencies visible. They clarify which context defines the meaning of shared data, how that meaning is communicated, and where responsibility lies when interpretations change. This clarity provides the foundation for treating shared data not as an accidental byproduct of systems, but as an intentionally designed capability.

Figure 13-1 provides a simple example of context map relations in the StreamForge data catalog. The figure shows three bounded contexts (rounded rectangles), the data products owned by each context (hexagons), and three representative context map relationships (arrows) between them. Each relationship has a specific relationship type that describes the relation between the two bounded contexts.

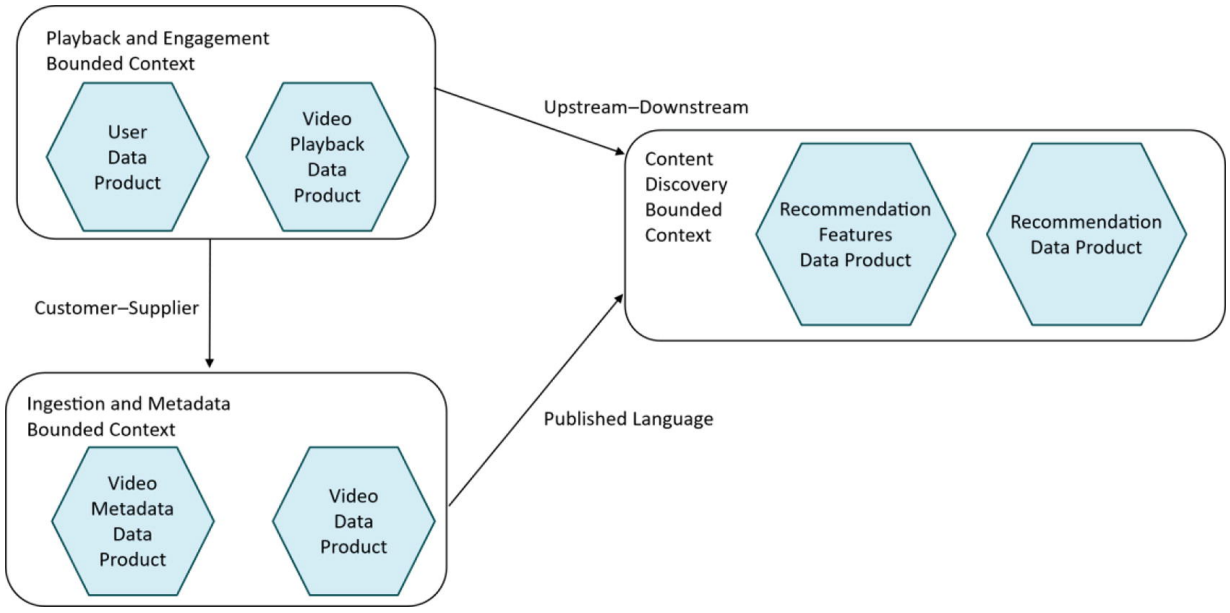


Figure 13-1 Bounded contexts, data products, and context map relationships

These relationships describe how bounded contexts coordinate meaning. In an upstream–downstream relationship, one bounded context produces data, events, APIs, or definitions that another context consumes. Changes in the upstream context may therefore affect downstream data products, even when the downstream context maintains its own internal model.

A customer–supplier relationship is a more explicit version of this dependency. The supplier context owns the data product, but the customer context has recognized requirements that should influence its evolution. This relationship is especially important for data contracts, SLAs, versioning, and compatibility commitments.

A published language relationship defines a stable language for integration. Instead of exposing every detail of its internal model, one bounded context publishes a vocabulary, schema, API, ontology module, or data contract that other contexts can use.

These relationships as well as other common DDD context relations are summarized in Table 13-1. They are particularly important for managing polysemes. Concepts such as *Video, User, or Recommendation* may appear in several bounded contexts, but each context may define them differently. The context map makes those differences explicit and clarifies whether another context should conform to an upstream definition, negotiate changes with a supplier, consume a published language, or translate the concept into its own model.

Table 13-1 Example Context Map Relationship Types

Figure 13-2 shows the knowledge graph implementation of the concepts in Figure 13-1. This (and all graph visualizations in this chapter) is from the Semantic Knowledge Graph https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/knowledge_graph/streamforge_data_catalog_V1.ttl.

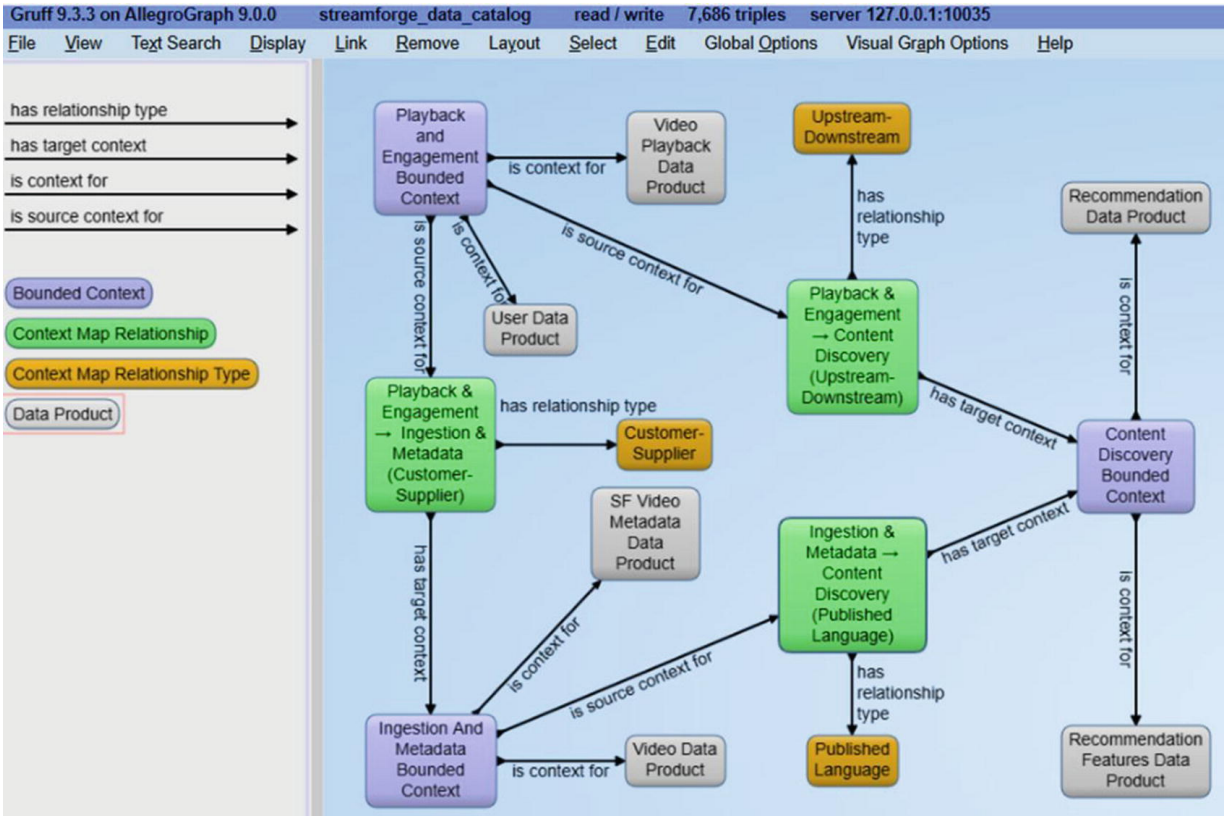


Figure 13-2 Knowledge Graph entities that implement the DDD concepts in Figure 13-1

The next section shows how context map relationships and bounded contexts can manage change to polysemes.

Polysemes and Domain Concepts Across Bounded Contexts

A bounded context does more than group software services and data products. It also scopes the meaning of domain concepts. In Figure 13-1, the StreamForge catalog includes three bounded contexts: `Ingestion_and_Metadata`, `Playback_and_Engagement`, and `Content_Discovery`. Each context uses terms that are meaningful across the company, but those terms do not always mean exactly the same thing in every context.

A common example is the term Video. At first glance, it may seem obvious what a video is. However, the meaning shifts depending on the bounded context. In the `Ingestion_and_Metadata` context, a video is primarily a managed media asset prepared for cataloging, storage, encoding, and metadata enrichment. In the

`Playback_and_Engagement` context, a video is a playable content item experienced by a user during a session. In the `Content_Discovery` context, a video is a candidate item for ranking, personalization, similarity comparison, or recommendation generation.

This is an example of a polyseme: the same term has related but context-specific meanings (Dehghani, 2022). Table 13-2 summarizes the three StreamForge meanings of the polyseme Video.

Table 13-2 The Three StreamForge Meanings of the Video Polyseme

A tempting modeling approach would be to create a single enterprise-wide `Video` class and then make the context-specific meanings subclasses of that class. That approach is sometimes appropriate when the relationship is truly one of logical specialization. However, it is often misleading for polysemes. In OWL, subclassing means logical subsumption: every instance of the subclass is also an instance of the superclass, and any constraints on the superclass apply to all subclasses.

That is an example of the old way of managing enterprise data by one enterprise-wide data model. This approach is usually an impediment to rapid, agile change. Data Mesh and DDD treat the concept of an enterprise data model as an anti-pattern (Dehghani, 2022).

For example, if the enterprise-level `Video` class required a playback URL or an availability status, then ingestion-stage media assets might be forced into constraints that do not yet apply. Conversely, if the enterprise-level definition were kept so weak that it imposed no meaningful constraints, it would function as an organizational bucket rather than an actual ontology class which is a conceptual modeling anti-pattern.

For this reason, the StreamForge catalog models these meanings as SKOS-style domain concepts rather than as subclasses of a single OWL class. The general `Video` concept is connected to the three context-

specific concepts using `skos:narrower` and `skos:broader`, not `rdfs:subClassOf`. This represents conceptual grouping without asserting logical subsumption. Figure 13-3 shows a graph view of the three definitions of the polyseme concept video and the bounded contexts each is associated with. Table 13-3 shows the same information in table form.

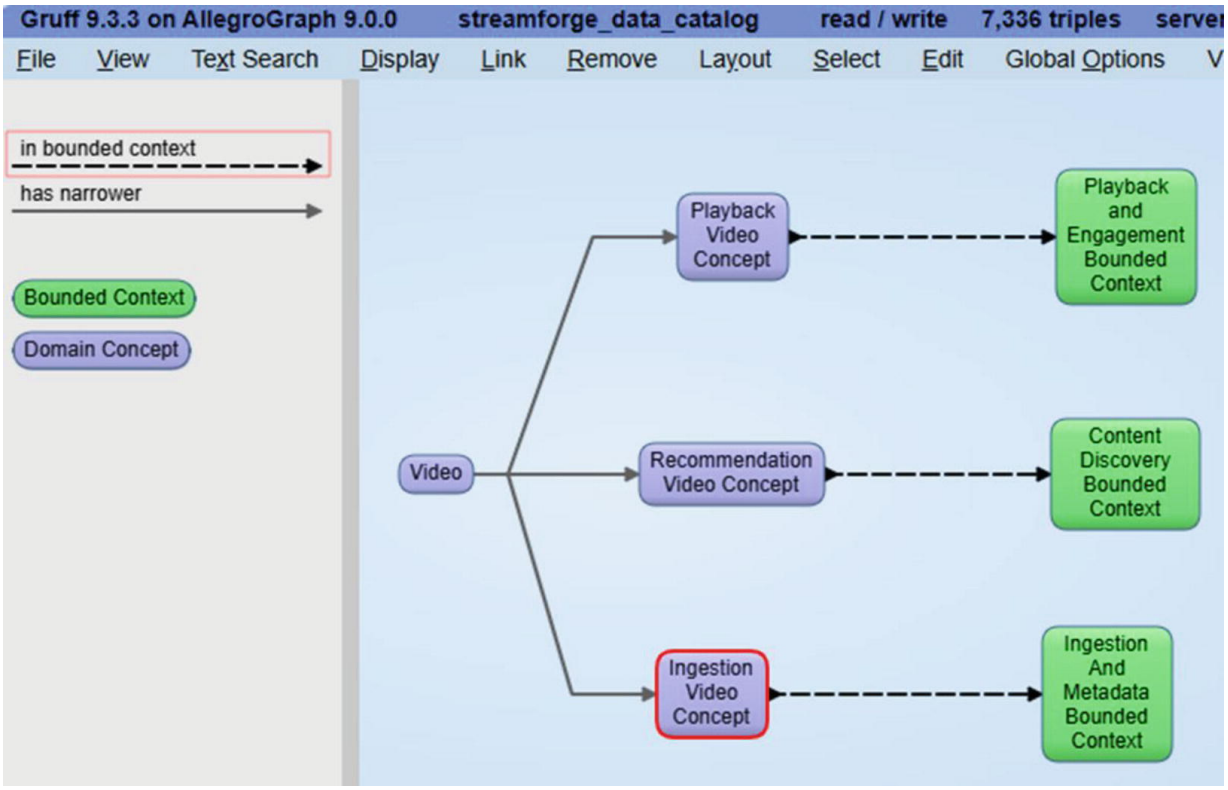


Figure 13-3 Modeling polyseme domain concepts with SKOS

Table 13-3 Video Concepts, Labels, and Bounded Contexts

This pattern preserves two important facts at the same time. First, users in all three contexts may naturally say “Video.” Second, the semantic model does not pretend that the term has one identical meaning everywhere. The concept hierarchy groups related meanings, while the bounded context relation identifies where each meaning applies.

The context map becomes especially important when a domain concept changes. Suppose the Playback and Engagement team changes its local meaning of `Video` by adding an availability status. Playback services now distinguish videos that are playable, blocked, expired, or region restricted. This is not merely a new attribute on a data structure. It changes the assumptions that other systems may make about what counts as a playable video.

The semantic catalog represents this as a change to the Playback Video Concept within the Playback and Engagement bounded context. The most important parts of the graph that model this change are shown in Figure 13-4.

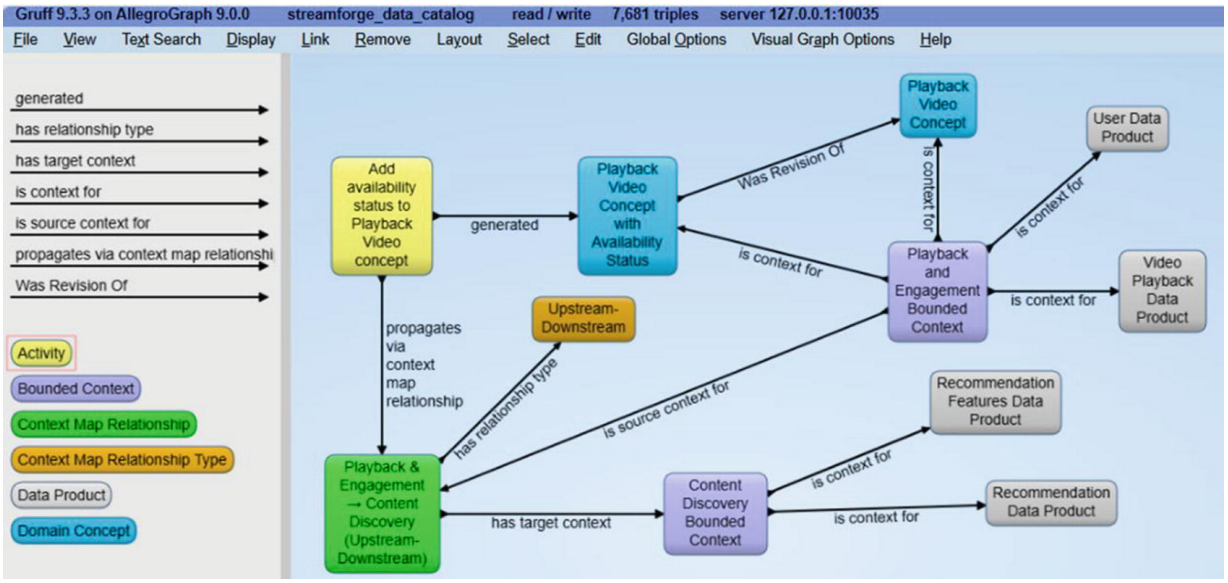


Figure 13-4 Tracking changes to domain concepts in the knowledge graph

The change may also be connected to the data products, ports, and implementation artifacts that operationalize that concept. For example, a port implementation that publishes playback events may need to include the new availability status, or at least document how availability is represented. This is shown in Figure 13-5.

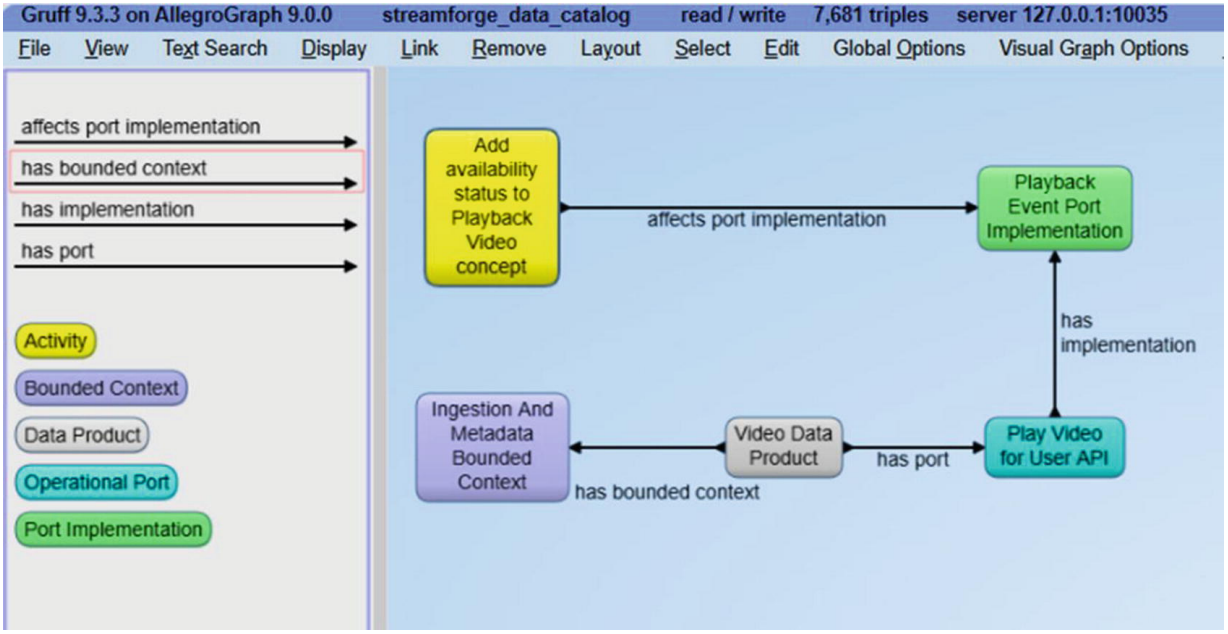


Figure 13-5 Tracking changes to port implementation

The important point is that the change does not propagate uniformly to every other bounded context. Its impact depends on the context map relationship. If content discovery is downstream from Playback and Engagement, the availability-status change may affect recommendation logic. A recommendation data product should not recommend videos that are blocked, expired, or unavailable in the viewer's region. The content discovery team may therefore need to review its recommendation video concept, its recommendation features, and any data products that consume playback events.

In a customer–supplier relationship, the impact is different. The supplier context still owns the data product and its semantics, but the customer context has recognized requirements that should influence the evolution of the product. A change such as availability status may therefore require negotiation, compatibility analysis, versioning, or updates to a data contract.

In a published-language relationship, the rule is different again. Internal changes inside the publishing context do not affect other bounded contexts. Only changes to the published vocabulary are significant. There are many ways to describe a published vocabulary.

For example, it can be a schema, ontology model, interface definition, or data contract. The important thing is it defines a contract between two bounded contexts and guides changes to terms across context boundaries. The published vocabulary defines the public model that two or more bounded contexts share. In this example, if the addition of an availability status doesn't change the published language that the Playback and Engagement context publishes to other contexts, then those contexts will not need to change.

An anti-corruption layer represents a still different case. The consuming context should not directly absorb the upstream model change. Instead, the translation layer must be reviewed to determine whether the changed upstream concept should be mapped, ignored, transformed, or isolated from the downstream model.

In this way, the context map acts as an impact-analysis guide. The catalog can identify what changed, which bounded context owns the changed concept, which data products and ports implement it, and which other bounded contexts are connected to it. The context map then tells the organization what kind of response is appropriate: downstream review, customer–supplier negotiation, published-contract versioning, or translation-layer adjustment.

In the next section, we examine the relation between bounded contexts and data products.

From Bounded Contexts to Data Products

Once bounded contexts and their relationships are made explicit, a natural question arises: how should data be shared across those boundaries? In most organizations, this question is answered implicitly and inconsistently. Data produced within one domain is copied into shared databases, exposed through ad hoc APIs, or extracted through batch pipelines whose semantics are only partially understood by their consumers. Over time, these mechanisms accumulate technical debt in the form of hidden dependencies that erode the boundaries that bounded contexts were meant to protect.

Data products address this problem by providing a principled way for bounded contexts to expose data to others. ***A data product is the data-facing counterpart to a microservice:*** a managed, versioned capability that is owned by a specific domain and designed for consumption by others. Where a bounded context defines the boundaries of specific domains, a data product defines how data in each context is used and shared.

A key implication of this framing (which we model in the ontology as an axiom on the `Data_Product` class) is that every data product has exactly one single owning bounded context. That context is responsible not only for producing the data, but for defining its semantics, documenting its intended use, and managing its evolution over time. Data consumers may depend on a data product, but they do not define its meaning. This separation of *ownership* from *consumption* mirrors the relationship between microservices and their clients, and it is essential for maintaining clear accountability.

It is important to distinguish data products from several related but incomplete concepts. A data product is not simply a dataset, even if it is implemented using a table, file, or stream. Datasets describe storage artifacts; data products describe *capabilities*. Nor is a data product equivalent to a service endpoint. While services typically expose behavior, data products expose data along with semantics, metadata, and constraints that make that data usable beyond its original context.

This distinction becomes clearer when considering evolution. Data products are designed to evolve over time without breaking the contracts that downstream data consumers depend on. They may introduce new attributes, support additional access mechanisms, or deprecate obsolete representations. These changes are managed through explicit contracts, versioning, and communication between producers and consumers. A data product is a long-lived product, not a transient data extract.

From a modeling perspective, data products provide a concrete anchor point for many of the architectural concerns discussed earlier in this chapter. Ownership is explicit in data products. Integration is

abstractly described by context maps and is realized through well-defined data ports. Governance concerns such as quality, lineage, and access control can be attached directly to data products as axioms and SHACL constraints.

Semantic models play a crucial role in making these properties explicit. By modeling data products as first-class entities within a knowledge graph, organizations can represent not only the data, but the concepts it embodies, the governance and metadata constraints it must conform to, the context and teams that own and use it, the relationships it has with other products, and additional metadata about how and when it was created and modified. This allows architectural assumptions about data sharing to be inspected, queried, and reasoned about in ways that are difficult to achieve with purely procedural or document-based approaches.

Crucially, treating data as a product does not imply a particular deployment architecture. Data products can coexist with centralized warehouses, data fabrics, data mesh or (what is usually the case in the real world) hybrid platforms. What matters is not where the data resides, but how responsibility for its meaning, quality, and evolution is assigned. In this sense, data products apply the lessons of Domain-Driven Design to data, without prescribing how an organization must structure its infrastructure.

Table 13-4 summarizes these relations and how to model them in OWL.

Table 13-4 Context Relations Between Bounded Contexts, Data Products, and Data Ports

In the next section, we turn to the role of the data catalog. Data products define what is owned and shared; catalogs provide the mechanisms by which those products are discovered, understood, and connected. Semantic Knowledge Graphs provide the connective tissue that allows these two concepts to reinforce one another. Without product semantics, catalogs degenerate into inventories of technical artifacts. Without catalogs, data products remain invisible.

The Role of the Data Catalog

As organizations adopt data products, the question of discovery becomes unavoidable. If data is owned and exposed by multiple bounded contexts, how do potential consumers find what exists, understand what it means, and determine whether it is appropriate for their use? The data catalog addresses this problem by providing a shared point of reference for available data capabilities.

However, the role of a data catalog is often misunderstood. In many organizations, catalogs have been treated as inventories of technical artifacts: tables, files, schemas, dashboards, or pipelines. While such information is necessary, it is not sufficient. Catalogs that focus exclusively on infrastructure details tend to become either overwhelming or irrelevant, especially as the number of assets grows.

A core limitation of traditional catalogs is that they lack semantics (Olesen-Bagneux, 2023). Without an explicit notion of ownership, purpose, and contractual guarantees, catalog entries become disconnected fragments of technical metadata. Users may be able to locate a dataset, but they cannot answer essential questions: Who is responsible for this data? What does it represent? Where did it come from? How stable is it? How should it be used and how may it not be used?

Data products provide the missing semantic anchor. When data is organized around products rather than raw artifacts, the catalog's role shifts from documentation to discovery. Instead of listing tables or files in isolation, the catalog presents data products as coherent capabilities: named, owned, versioned, and described in terms of the domain concepts they expose. Technical details such as storage formats, access mechanisms, and implementations become supporting information rather than the primary organizing principle.

From this perspective, a data catalog does not define what a data product is; it makes data products visible and usable. The semantics of a data product, its meaning, boundaries, and constraints, are defined by the owning bounded context. The catalog provides a structured way to

publish that information and make it discoverable across the organization.

This distinction helps explain why many catalog initiatives struggle. Catalogs built without an underlying semantic model tend to accumulate entries faster than they accumulate understanding. Conversely, data products without a catalog remain effectively invisible outside their immediate domain. The two concepts reinforce one another: data products provide the semantics that make catalogs useful, and catalogs provide the visibility that makes data products valuable.

Semantic Knowledge Graphs offer a natural foundation for unifying these concerns. By representing data products, bounded contexts, and catalog entries within a shared graph, organizations can connect descriptive metadata, domain semantics, and technical details without collapsing them into a single global schema. Relationships such as ownership, dependency, and usage can be expressed explicitly rather than inferred indirectly.

This approach also prepares catalogs for emerging consumption patterns. As analytic tools, automated pipelines, and large language models increasingly rely on data and metadata to guide behavior, the quality, lineage, and structure of data become critical. Catalogs that expose only technical details are difficult for both humans and machines to interpret. Catalogs grounded in product semantics and explicit domain models are far more amenable to automation, reasoning, and integration.

In the next section, we bring these ideas together by examining how data products, data catalogs, and Semantic Knowledge Graphs form a coherent architectural pattern. This pattern not only supports traditional analytics and integration but also provides a foundation for systems that combine deductive reasoning with inductive, data-driven approaches.

Data Products, Catalogs, and Semantic Knowledge Graphs

The concepts introduced in this chapter describe a coherent way of structuring data-intensive systems around *meaning, ownership, and evolution*. Domain-Driven Design provides the conceptual foundation by defining bounded contexts as the units within which meaning is stable and shared. Context maps make the relationships among those contexts explicit, clarifying how integration occurs and where translation is required. Data products implement these ideas by defining how data is exposed across boundaries in a controlled, accountable manner.

Data catalogs complete this picture by enabling discovery. They provide the mechanisms by which data products become visible beyond their contexts, allowing potential consumers to understand what exists, who owns it, and how it is intended to be used. Catalogs do not replace domain models or product definitions; they depend on them. Without explicit product semantics, catalogs degenerate into inventories of technical artifacts. Without catalogs, data products remain isolated and underutilized.

Semantic Knowledge Graphs offer a unifying substrate for these concerns. By representing bounded contexts, data products, catalog entries, and their relationships within a shared graph, organizations can preserve local autonomy while still supporting global reasoning. Semantic models make ownership explicit, encode meaning in a machine-readable form, and allow architectural assumptions to be automatically validated and inspected via graph query and transformation languages such as SPARQL. The flexible architecture of SPARQL, where one can search subjects, predicates, and/or objects, is ideally suited to this type of search. What's more, architectures such as retrieval-augmented generation (RAG) enable the catalog to also be utilized using natural language via a large language model (LLM) as we will show later in this chapter.

This approach accommodates the realities of modern data-driven systems. Data products may be implemented using event streams, analytic warehouses, APIs, or combinations of these and other technologies. Catalog information may be enriched incrementally,

evolve over time, or be consumed programmatically by automated systems. Semantic Knowledge Graphs do not require these choices to be fixed in advance. They provide a flexible framework within which such variability can be represented without losing conceptual clarity.

In the remainder of this chapter, we extend the concrete example described so far. Using the StreamForge domain, we illustrate the natural synergy between Semantic Knowledge Graphs and data catalogs. In addition, we'll demonstrate how the data catalog can be turbo-charged using a retrieval-augmented generation (RAG) architecture that allows the user to not just query what is in the catalog but reason about the impact of potential changes. The goal is not to present a complete or prescriptive solution, but to demonstrate how the concepts discussed so far can be realized in practice and how the combination of semantic technology with LLMs provides unprecedented capabilities to define, use, and manage enterprise data.

The StreamForge Example: Data Products in Context

StreamForge is a simplified, hypothetical video-streaming platform used to illustrate how Semantic Knowledge Graphs can support real-world architectural concerns without overwhelming detail. In this section, StreamForge serves as an example of how bounded contexts, context maps, and data products can be modeled in a data catalog using OWL and SHACL and queried using SPARQL and a large language model. The approach described here is based on work of mine, designing a Semantic Knowledge Graph data catalog for a large-scale, consumer-facing client integrated with other work where I've developed retrieval-augmented generation (RAG) systems for various domains.

With this background, we can now use StreamForge to show how bounded contexts, context map relationships, and data products fit together in a semantic data catalog. In StreamForge, bounded contexts such as ingestion and metadata management, content discovery, and

creator analytics each define their own model of the domain. Terms such as video, playback, and engagement may appear in more than one context, but their meanings, constraints, and operational responsibilities differ depending on how each context uses them. The purpose of the data catalog is not to collapse these differences into a single enterprise-wide definition, but to make the contextual boundaries and relationships explicit.

Context map relationships describe how these bounded contexts depend on and communicate with one another. Once a data product is defined in this way, it can support multiple access patterns. The same conceptual product might be made available as an event stream for near-real-time processing, as a queryable analytic view for reporting, or as a precomputed summary for user-facing applications. These variations represent different implementations of the same product contract rather than different products. The contract defines what is offered; the implementations define how it is delivered.

From a modeling perspective, these distinctions are made explicit. Data products are modeled as first-class entities associated with the bounded contexts that own them. Data product ports describe the capabilities offered to consumers (note consumers here primarily refers to other business teams within the same organization), while individual implementations capture technology-specific details such as access mechanisms or underlying platforms. This separation allows data products to evolve technologically without changing their conceptual meaning or breaking existing contracts.

Catalog-style metadata is layered on top of this structure. The flexibility of SPARQL means that data products can be discovered by searching across many different dimensions including name, domain, owner, or intended use. Relationships among products, contexts, and implementations can be traversed and queried. Rather than cataloging raw tables or streams in isolation, the catalog reflects the conceptual organization of the system, grounded in domain ownership and explicit semantics.

This example also highlights an important theme that recurs throughout the book: semantic modeling is most effective when it supports architectural intent rather than attempting to replace it. The goal is not to model every detail, but to model the right details that clarify responsibility, meaning, and interaction across system boundaries. This is one of the most important principles for modeling in general. People who attempt to "model the world" fail to understand the essence of what a model is. A model is an abstraction that abstracts away some details so that we can focus on the details that matter for the specific problems we need to solve.

Visualizing the StreamForge Knowledge Graph

This section uses a small number of diagrams generated directly from the StreamForge knowledge graph using SPARQL and the AllegroGraph Gruff visualization tool. Each figure visualizes entities and relationships that are explicitly modeled in the ontology and populated in the example graph.

The goal is to show how Domain-Driven Design concepts, data products, and catalog metadata coexist within a single semantic model. These figures were generated by querying and visualizing instances and classes already present in the knowledge graph using Gruff.

Core Domain Concepts As a Semantic Graph

Data products are higher-level constructs that package access to these domain concepts through well-defined ports. A video data product, for example, may expose ports such as `Get_Video_Metadata`, `Get_Trending_Videos`, `or Play_Video_For_User_API`, each of which provides a specific view or interaction over the underlying domain model.

This separation allows the same domain entities to support both operational and analytic use cases, without duplicating models or flattening semantics into a single schema. This demonstrates

- The ubiquitous language of the domain
- Conceptual entities independent of implementation

- A shared vocabulary that spans multiple bounded contexts

Data Products Within a Bounded Context

Figure 13-6 shows a Gruff visualization with two `Bounded_Contexts`, the `Data_Products` associated with them, as well as other properties and objects such as ports.

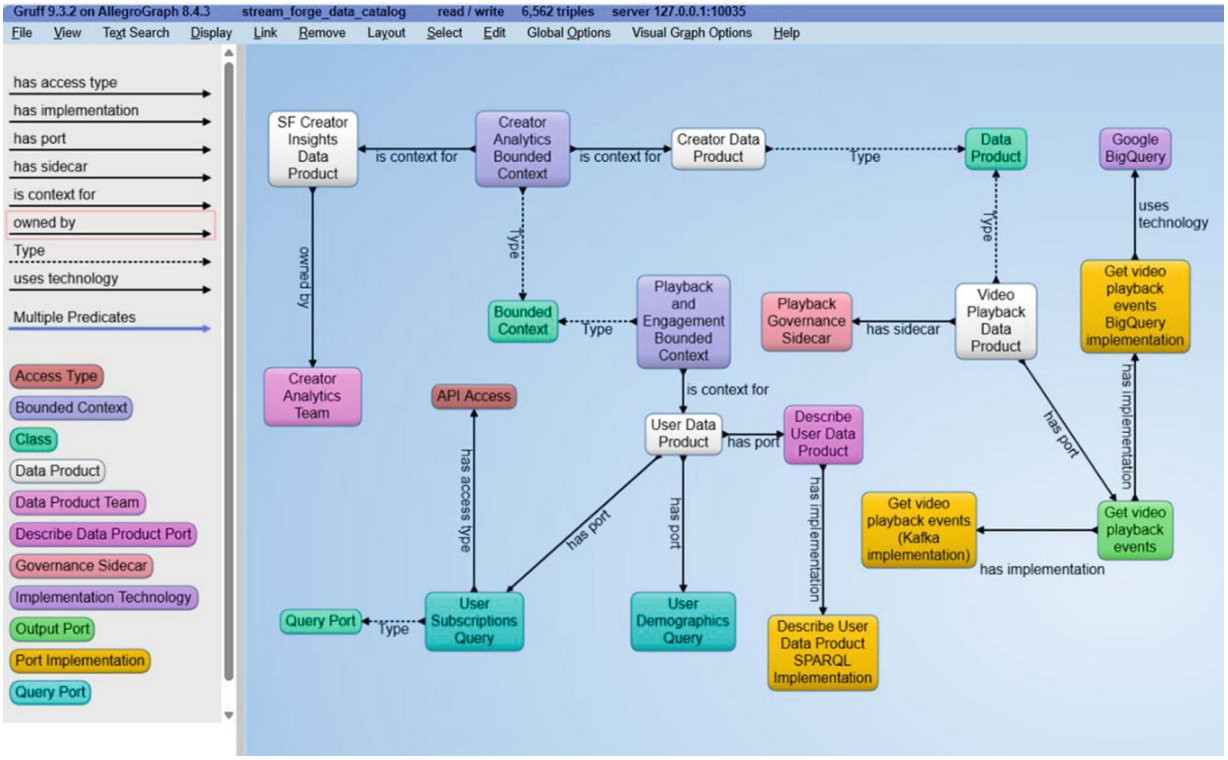


Figure 13-6 Data products, bounded contexts, ports, and implementations

This figure makes several architectural commitments explicit:

- Data products are owned by exactly one bounded context.
- Products expose capabilities through ports, not internal structure.
- Multiple access patterns coexist without fragmenting product identity.

The diagram directly reflects the ontology structure: `Data_Product` and `Data_Product_Port` are classes and specific products and ports are instances.

From Conceptual Product to Implementation

Figure 13-7 focuses on a data product port, its implementation, technology, and access type. This figure illustrates a crucial distinction: the contract of a data product is separate from its technical realization. Because implementations are modeled explicitly, they can change without altering the meaning of the product or breaking dependent systems. This mirrors the way microservices separate interfaces from internal code and data.

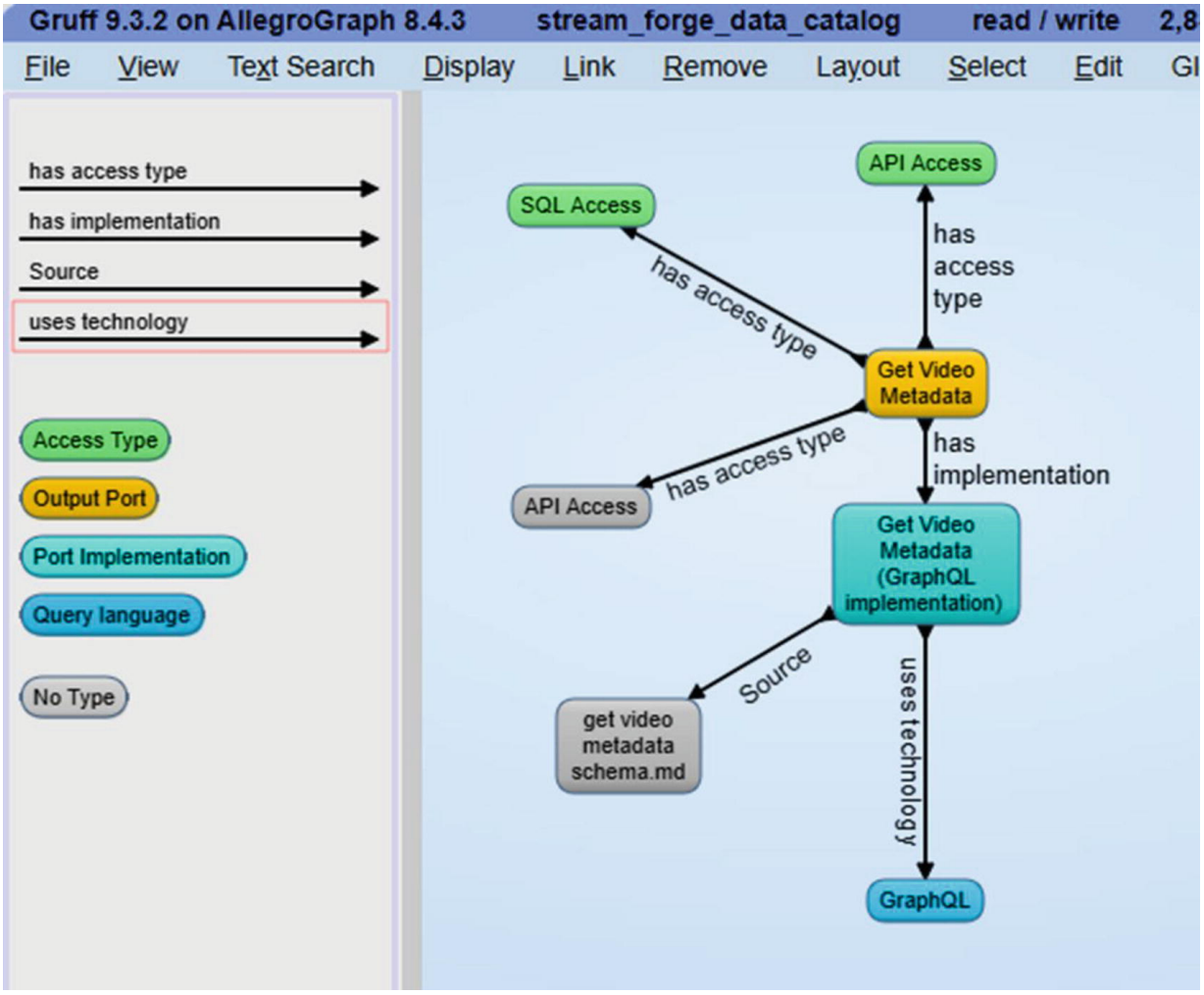


Figure 13-7 Gruff visualization of a data product port, its implementation, technology, and access types

Together, these models show how Domain-Driven Design concepts, data products, and catalog metadata coexist within a single Semantic

Knowledge Graph. The graph is not merely a catalog or metadata store; it functions as an integration layer that connects domain meaning, ownership, governance, and technical realization. By combining standards such as DCAT and PROV-O with explicit modeling of bounded contexts and data products, architectural intent becomes inspectable and explicit.

Implementing Data Product Ports with REST and OpenAPI

Up to this point, data product ports have been described at a conceptual level: they define the interaction points through which data is exposed to consumers. A natural next question is how these ports are realized in practice. In modern systems, this typically involves one or more concrete interface technologies such as REST APIs, event streams, or query services such as SQL and SPARQL.

This section goes into some depth on REST-based ports described using the OpenAPI specification (Andrews et al., 2025). OpenAPI is one of the most common standards used to define REST APIs. The goal is not to prescribe REST as the primary mechanism for exposing data products, but to provide a specific example of how a widely adopted interface description can be aligned with the semantic model introduced earlier in this chapter.

From a Domain-Driven Design perspective, a port corresponds to a well-defined contract between a provider and its consumers. OpenAPI provides a standard way to make that contract explicit. It defines the available operations, the structure of requests and responses, and the protocols by which they are accessed. In this sense, OpenAPI plays a role similar to an Interface Definition Language (IDL).

Standards such as OpenAPI make it possible to define the ports through which a data catalog exposes its resources. The ontology defines the meaning of the data being exposed, while the OpenAPI specification defines how that data can be accessed. In addition, the specific interfaces and technologies are also modeled in the knowledge

graph. Other examples of implementation technology that are modeled in the StreamForge Data Catalog include Google BigQuery, SQL, SPARQL and Kafka Topics. In each case, an IRI is the value of a property. In this simplified example, each of the IRIs points to a document in the GitHub repository that stands for the integration point. In an actual data catalog, it would be the entry point to the specific technology and data product.

This distinction is critical. Without an explicit semantic model, an API description is limited to structural information: field names, data types, and endpoints. With a semantic model, those same elements can be interpreted in terms of domain concepts, relationships, and constraints that are consistent across data products and bounded contexts.

This also facilitates one of the crucial requirements for a data product: self-service and discoverability. In an implemented system, links on discovery ports would launch SPARQL queries, as shown below.

Mapping Ports to OpenAPI Interfaces

The relationship between the semantic model and a REST interface can be understood as a layering:

- A *data product* defines a domain-owned capability.
- A *data product port* defines an abstract interaction point.
- *Constraints on a data product port* such as that it must use a REST architecture are modeled as axioms on the class that describes the port.
- A *port implementation* defines a concrete interface using a specific technology such as OpenAPI REST.

In OWL terms, this corresponds directly to the patterns introduced in Table 13-4:

```
Data_Product → Data_Product_Port via has_port
Data_Product_Port → Port_Implementation via
has_implementation
```

An OpenAPI specification is one form of `Port_Implementation`. It describes how a particular port is realized as a set of HTTP endpoints. To keep the explanation simple, we're glossing over some details. For example, there are various subclasses of the `Data_Product_Port` class (e.g., `Analytic_Data_Port` and `Platform_Port`) as well as sub-properties of `has_port` corresponding to these subclasses. The reader is encouraged to load the StreamForge knowledge graph into Protégé and to utilize graphical browsers such as Gruff to do their own SPARQL queries and graph visualizations of the Data Catalog.

For example, consider a port for the `Video_Playback_Data_Product` in the `Playback_and_Engagement_Bounded_Context`. Conceptually, this port might be described as:

Provides access to curated playback events for downstream analytic and operational use

This conceptual description is part of the semantic model. It may be represented using properties such as `rdfs:label` and `skos:definition`. The OpenAPI specification then defines how a consumer interacts with this port:

- A GET /playback-events endpoint to retrieve events
- Query parameters for filtering by time range or video identifier
- A response schema describing the structure of returned events

The OpenAPI document does not define what a playback event means. That meaning is defined by the ontology. Instead, it defines how that concept is accessed over a network.

Separating Semantics from Interface Structure

This separation between semantics and interface structure is one of the main advantages of combining semantic models with API specifications.

In many systems, API schemas become the de facto source of truth for meaning. Field names and data types are treated as if they fully

describe the domain. This works for small systems, but it breaks down as systems grow and concepts are reused across contexts. The same term may be used with slightly different meanings, or different terms may be used for the same concept.

By contrast, when semantics are defined in an ontology:

- API schemas can be interpreted consistently across systems.
- Changes to interface structure do not necessarily imply changes to meaning.
- Multiple interfaces (REST, event streams, query services) can expose the same underlying concepts.

In this model, the OpenAPI specification is an example projection of the semantic model into a particular interaction style.

Multiple Implementations of a Single Port

A single data product port may have multiple implementations. For example, the Video Playback data product might simultaneously expose

- A REST API described with OpenAPI for on-demand queries
- A Kafka topic for streaming events
- A batch export to a data warehouse

These are not separate data products. They are different implementations of the same product contract. Each implementation serves a different class of consumers but is governed by the same semantics and ownership.

This distinction is important for both architecture and governance. It allows systems to evolve technologically without fragmenting the conceptual model. A new interface can be added without redefining the data product, and an existing interface can be deprecated without losing the underlying capability.

Ports As First-Class Architectural Elements

By modeling data product ports explicitly, the architecture makes interfaces visible and queryable. Instead of being scattered across code

repositories or documentation, ports become part of the knowledge graph itself.

This enables questions such as

- Which data products expose REST interfaces?
- Which ports depend on a given bounded context?
- What implementations exist for a given port?
- Which consumers depend on a particular interface?

These questions are difficult to answer in traditional architectures, where interfaces are implicit or distributed across multiple systems. In a Semantic Knowledge Graph, they become straightforward graph queries using SPARQL, Graph RAG, or keyword search. The various search options will be described in more detail below.

This approach aligns implementation with the ubiquitous language of the domain. The same concepts that appear in the ontology are reflected in data products, ports, and interfaces. This reduces the translation gap between domain understanding and system behavior, which is the central goal of Domain-Driven Design.

In the next section, we move from visualization to interaction.

Querying the Data Product Catalog

Rather than manually browsing documentation or inspecting schemas, users and tools interact with the catalog through platform ports that expose discovery and inspection capabilities. These ports are read-only, cross-cutting by design, and intentionally span bounded contexts. In this section, we query the catalog using SPARQL. We will show more sophisticated ways to interact with the data catalog via natural language in later sections.

The SPARQL queries shown in the following figures are exposed through a platform port, reflecting their role as system-level capabilities for discovering and reasoning about data products and bounded contexts rather than as domain-specific APIs. Run these queries with RDFS++ inference enabled.

The query in Listing 13-1 illustrates a canonical discovery operation: listing data products together with their bounded contexts, exposed ports, and supported access types. The SPARQL query represents a stable contract to explore the catalog. In this example, the visualization for some of the knowledge graph objects and links returned by this query is shown in Figure 13-8.

```
PREFIX dp: <https://www.michaeldebellis.com/dp/>
CONSTRUCT {
  # Derived edge for visualization and discovery
  ?bc dp:has_data_product?dp.
  # Core product structure
  ?dp   dp:has_port ?port.
  ?port dp:has_interface ?iface.
  ?iface dp:has_access_type ?accessType.}
WHERE {
  ?dp a dp:Data_Product ;
      dp:has_bounded_context ?bc ;
      dp:has_port ?port.
  ?port dp:has_interface ?iface.
  ?iface dp:has_access_type ?accessType.}
```

Listing 13-1 Product Discovery Within Bounded Contexts

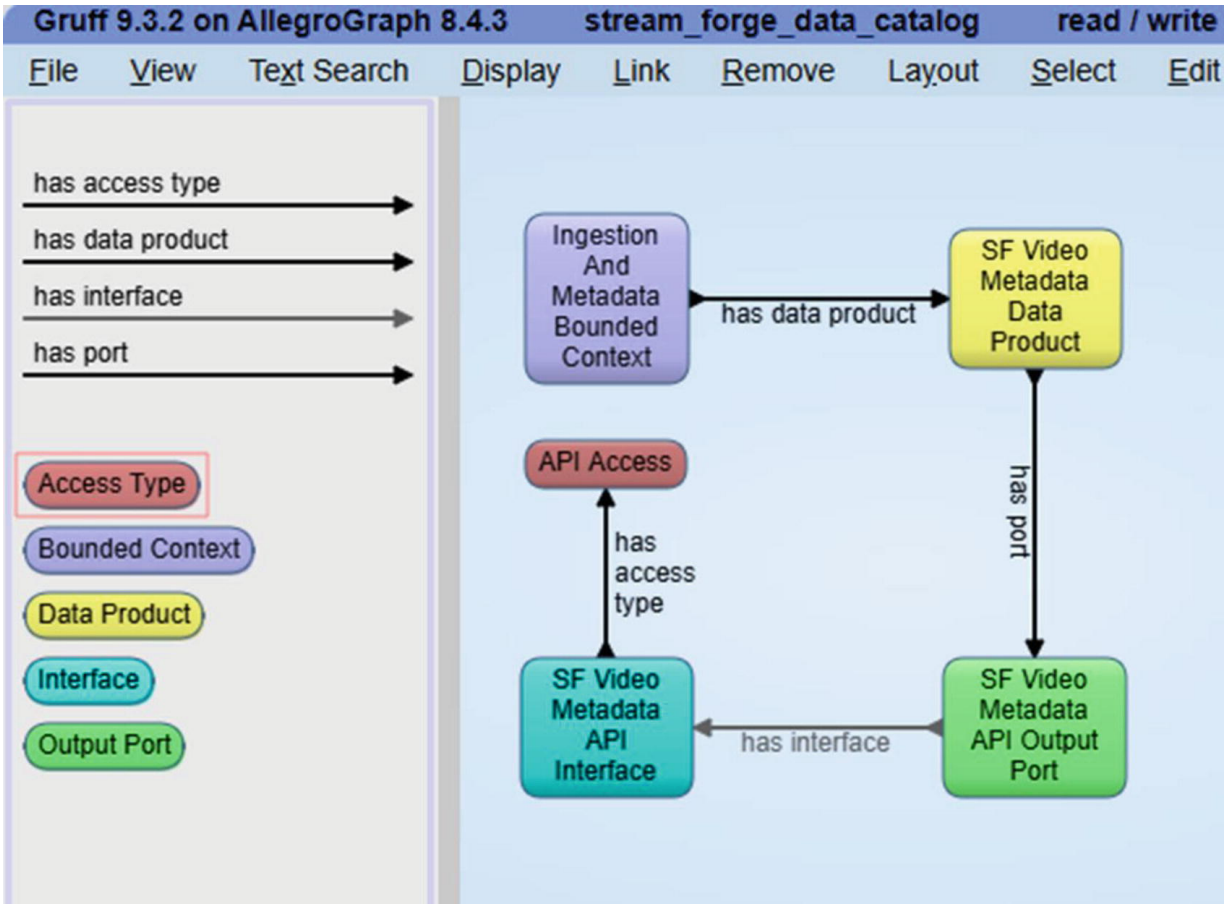


Figure 13-8 Gruff visualization of objects from SPARQL query in Listing 13-1

This query corresponds to a platform port of the data product catalog that supports discovery and inspection. Consumers do not need to know how products are stored or implemented; they interact with a stable semantic contract that exposes product boundaries, interfaces, and access modes.

The relationships involved in a Data Catalog are inherently graph-oriented rather than tabular. Ownership, dependency, lineage, composition, and evolution over time are all naturally expressed as networks of relationships. While tabular representations are effective for aggregations and reporting, they are less well suited to capturing how data products relate to one another, how responsibilities are assigned, or how changes propagate across boundaries.

Semantic Knowledge Graphs provide a natural representation for this kind of information. Rather than forcing relationships into fixed

schemas, they allow products, contexts, ports, and implementations to be modeled as first-class entities connected by explicit, queryable relationships.

Listing 13-2 is a SPARQL query that can examine bounded contexts and context map relations between them. Some of the results are shown in Figure 13-9.

```
PREFIX dp: <https://www.michaeldebellis.com/dp/>
SELECT DISTINCT *
WHERE {?bcs dp:is_source_context_for ?sc.
?sc dp:has_relationship_type ?rt;
dp:has_target_context ?bct.}
```

Listing 13-2 Architectural Intent Between Bounded Contexts

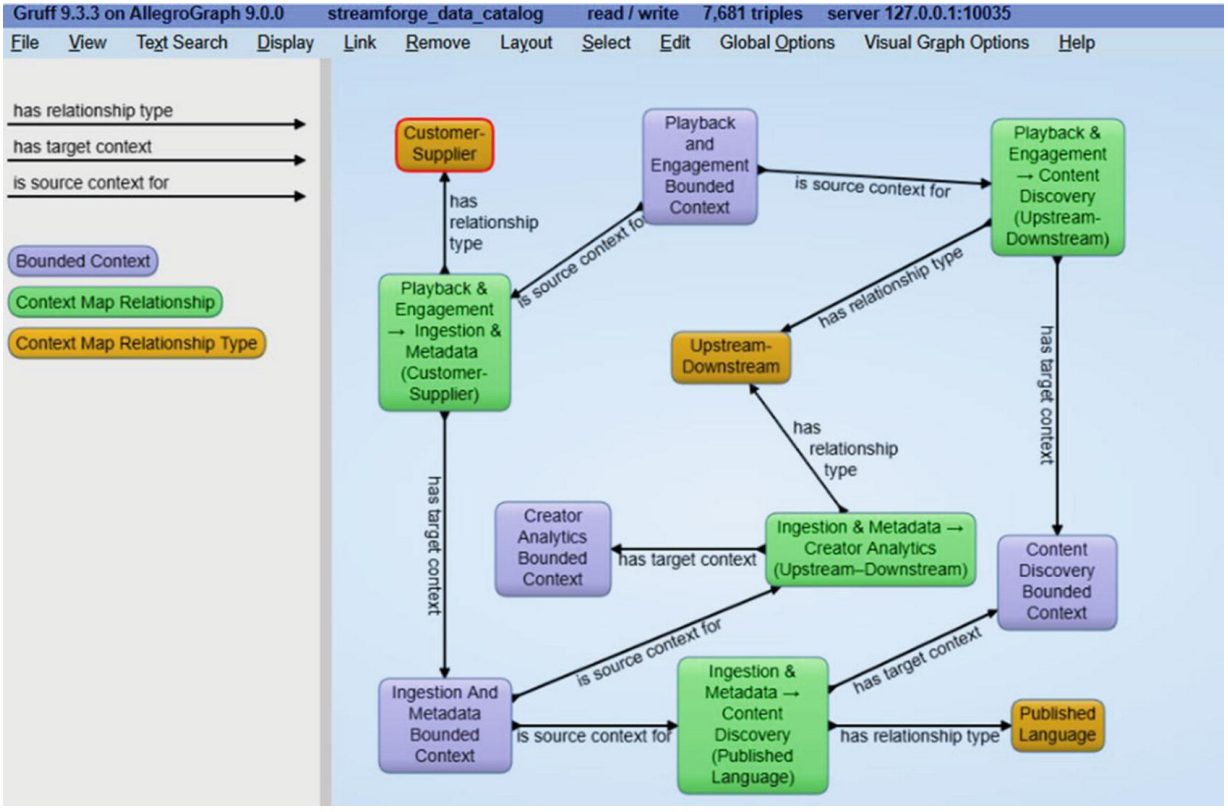


Figure 13-9 Architectural relations between bounded contexts from SPARQL query in Listing 13-2

Grounding Semantic Models in Production Analytics Platforms

The previous section operated at the level of the *conceptual model*, where data products are defined independently of implementation technology. We now shift to the *design model*, where those same products are realized using concrete analytics and streaming platforms.

Figure 13-10 illustrates several data products and Ports implemented using Google BigQuery and Kafka, showing how different technologies can back the same semantic contract. Notice that the same modeling pattern applies regardless of whether the product is backed by a warehouse or a streaming system. The ontology governs structure; the platform governs mechanics.

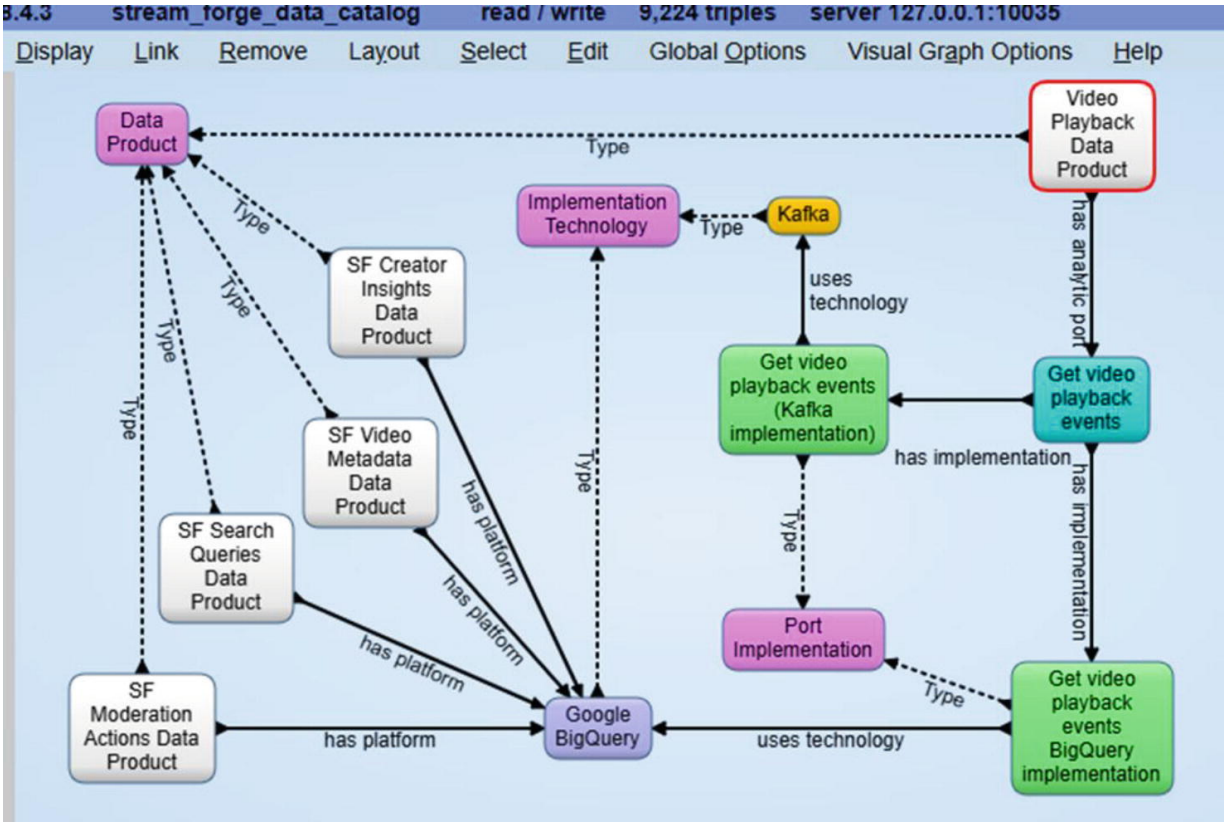


Figure 13-10 Modeling data platforms for data products and ports

In the StreamForge example, the `SF_Video_Metadata_Data_Product` is implemented using Google BigQuery. The knowledge graph provides a semantic layer that defines

ownership, meaning, and contracts above the physical storage and query engine.

The data product defines what is offered and under what guarantees. BigQuery is one platform used to implement an analytic port on that product. By modeling these concerns independently, implementations can evolve without changing the product's contract or breaking downstream consumers.

Figure 13-11 shows the kind of Video Playback Events query that StreamForge would implement in a platform such as Google BigQuery. In this prototype, BigQuery is not actually integrated with the knowledge graph; doing so would require infrastructure beyond the scope of this example. Instead, the query image is associated with the `Get_Video_Playback_Events_BigQuery_Implementation`, which implements the `Analytic_Data_Port` of the `Video_Playback_Data_Product`.

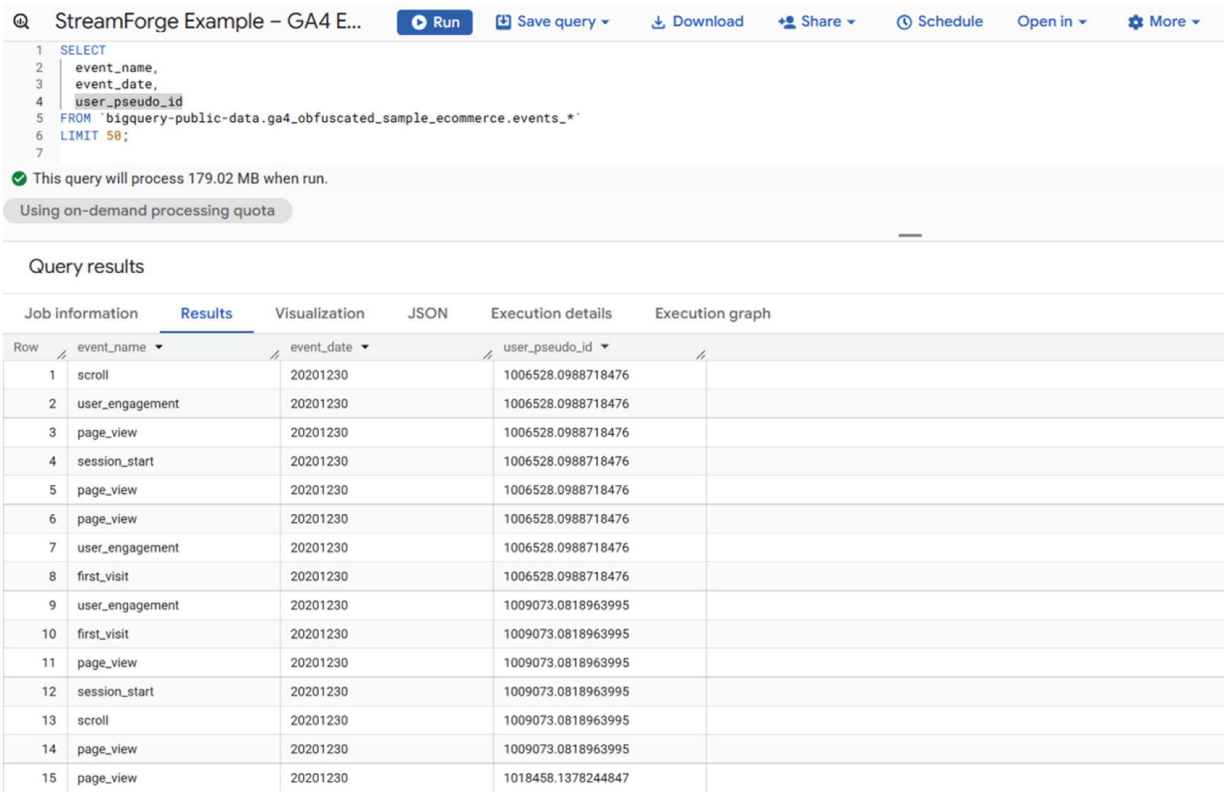


Figure 13-11 Example BigQuery query of user interaction events from the GA4 public dataset, illustrating a possible analytic port implementation

The key architectural point is that the implementation is represented as a first-class entity in the catalog. The IRI currently points to a static image for demonstration purposes, but in a production system, it would point to a live BigQuery console, an internal analytics portal, or another operational endpoint. The catalog provides a semantic entry point and context for various platforms.

Utilizing Large Language Models and SPARQL to Search and Leverage Enterprise Data

This section demonstrates how a Semantic Knowledge Graph data catalog can be combined with a large language model (LLM) to create a Graph RAG system: one that produces natural-language answers while remaining explicitly grounded in enterprise metadata, provenance, and governance structures.

The objective is to show how very modest additional effort can turn an existing data catalog knowledge graph into a powerful interactive system that supports discovery, explanation, and executive-level decision-making both via the SPARQL language as well as natural language via an LLM and RAG architecture. In later sections, a tool to easily generate Free Text Indexes will be shown as well as a new tool called GraphTalker that generates SPARQL from natural language. The point is that the combination of Semantic Knowledge Graphs, large language models, and SPARQL provides many unprecedented ways to build tools to manage and leverage enterprise data in ways that would have been dismissed as science fiction a few years ago.

Overview of the StreamForge Data Catalog Knowledge Graph

The StreamForge Data Catalog integrates multiple ways to understand enterprise systems, built on several reusable vocabulary ontologies:

- Data products and their constituent datasets
- Governance and ownership metadata
- Domain-Driven Design concepts such as bounded contexts

- Operational and analytic pipelines
- Provenance and lineage

The catalog is an integration layer: a single model that defines what data exists, how it is produced, who is responsible for it, and how it is used.

Reuse of Existing Ontologies and Standards

This example demonstrates one of the advantages of using the Semantic Web. It reuses established vocabularies for provenance, metadata, governance, and the basic data catalog, only adding new entities for concepts such as Data Products. The catalog reuses

- The W3C Data Catalog Vocabulary (DCAT) (Albertoni et al., 2024) for datasets and catalog concepts
- Dublin Core (The Dublin Core Metadata Initiative (DCMI), 2024) for titles, descriptions, and basic documentation metadata
- SKOS (W3C SKOS Working Group, 2005) for human-readable definitions, explanatory annotations, and metadata
- PROV-O (Prov W3C Working Group, 2013) for lineage, activities, agents, and derivation relationships
- Simplified Upper Level Ontology (SULO) (Michel Dumontier et al., 2025) for a standard upper model for basic concepts such as parts and subparts that aren't already included in PROV-O, SKOS, or Dublin Core
- Open Digital Rights Language (ODRL) (ODRL Community Group, 2018) provides a standardized vocabulary for expressing policies, permissions, and prohibitions, enabling governance rules to be formally defined and queried in the knowledge graph

These vocabularies serve two purposes:

1. *Semantic clarity and interoperability*: Since most experienced ontologists are familiar with these vocabularies, their use in this ontology is instantly comprehensible. Even for software architects who think OWL is only a nocturnal bird, these vocabularies still bring clarity because they were developed by people with deep

experience in each domain using terms that all good architects and developers will know.

2. *Reduced effort*: Building on these vocabularies allows us to leverage some of the best ideas on each topic with minimal effort and to save our effort for concepts like Data Products that currently have no standard vocabulary and can provide us competitive advantage.

With these ontologies as the foundation, I developed the following ontologies for the Data Catalog RAG system:

1. The `docs_ontology` is used to store documents that are broken down into chunks for the RAG system.
2. The `data_product_ontology` has the most important new schema (TBox) definitions such as `Data_Product`.
3. The `streamforge_ontology` has most of the ABox definition. That is, the other ontologies are meant to be general and reusable for different organizations. The `streamforge_ontology` has the data specific to the StreamForge hypothetical company such as StreamForge data products, teams, and context relationship maps.

The `docs_ontology` is used to store documents that are broken down into chunks. Recall from the discussion of RAG in **Chapter 9** that there is a "goldilocks zone" for the size of RAG text strings. RAG systems usually work best when documents are divided into chunks that are large enough to preserve a coherent idea but small enough to retrieve precisely. As a practical rule of thumb, this often means chunks of roughly 100 to 500 words. Usually a paragraph to a few paragraphs.

Text strings that are too large do not generate meaningful vectors. The embedding vectors are designed to be focused on a few concepts; large strings generate vectors that are too general. At the same time, strings that are too brief, a few words or a single sentence, also generate poor vectors (DeBellis, 2026). Thus, the primary purpose of

the `docs_ontology` is to take documents and divide them into "chunks" of the correct size to create meaningful vectors. There is a `has_chunk` property in the `docs_ontology` with domain `Document` and range `Document_Chunk`.

The `data_product_ontology` is the ontology that has the most important schema (TBox) definitions. These define classes and properties from Domain-Driven Design (DDD) and data products that aren't already available in DCAT or PROV-O. Some of the most important classes from this ontology are `Bounded_Context`, `Data_Product`, `Context_Map`, `Context_Map_Relationship` and `Data_Product_Port`. Some of the most important properties are `has_port`, `has_bounded_context` and `has_context_map_relationship`.

The `streamforge_ontology` ontology is almost exclusively ABox data. It defines the individuals, property values, axioms, and SHACL constraints that are unique to the hypothetical StreamForge company. In the figures in this chapter, virtually all of the individuals and property values displayed via Gruff and Protégé are in the `streamforge_ontology` ontology.

With this background, we're ready to look closer at the LLM/SKG synergies.

From Data Catalog Knowledge Graph to Graph RAG Catalog Chatbot

> *…within this decade, the data catalog will become the search engine for companies… search for anything, just like [on the web]. That technology is the data catalog…*
> —The Enterprise Data Catalog, Ole Olesen-Bagneux (Olesen-Bagneux, 2023)

One of the things that made me want to write this book was reading Olesen-Bagneux's excellent book on the Enterprise Data Catalog. Due to my work integrating Semantic Knowledge Graphs with LLMs to build

retrieval-augmented generation (RAG) systems, I thought that while Olesen-Bagneux's idea was absolutely correct, the technological landscape has shifted significantly since he published his book. The ultimate search is no longer keyword search but interacting with an LLM which can do reasoning as well as search. In addition, visualization tools such as Gruff provide graphical ways to view lineage and other knowledge in the data catalog.

Seeing how incredibly useful RAG systems were in various domains made me wonder: what could be accomplished if we took the code and documents for an enterprise and stored them in a catalog defined by a Semantic Knowledge Graph coupled with a user interface powered by an LLM? Quite a lot, I think. This is the ultimate destination of the concepts and technologies that have come before. The supporting materials in the GitHub repository for this book include a proof of concept Enterprise Data Catalog knowledge graph implemented with a retrieval-augmented generation (RAG) architecture. These materials are available in the directory: https://github.com/mdebellis/SemanticKG-Design/tree/main/data_product.

The AllegroGraph LLM tool described in **Chapter 9** makes creating a Graph RAG system very straightforward. In that chapter, we showed how the AllegroGraph WebView GUI enables the developer to provide AllegroGraph with a list of the specific properties that have strings or literals as values (e.g., `skos:definition, dp:embedding_note, doc:chunk_text`) and it creates vectors for each value of each property and associates each vector with the string and property value in the knowledge graph.

As described in **Chapter 9**, the user creates a prompt, and a vector is created for it. That vector is utilized to find vector embeddings whose similarity to the prompt meets or exceeds the minimum similarity score and the text strings that matched are sent to the LLM via another API. The LLM uses this input to generate a natural language response. In AllegroGraph, this runtime process is handled by a SPARQL query. A parameterized SPARQL Python function generates the complete query.

The template is a string that is a SPARQL query that utilizes the AllegroGraph "magic properties" related to LLMs (Franz Inc., 2024). The parameters that are passed and used to generate a new SPARQL query are the user's prompt and parameters that define the minimum similarity score and the maximum number of matching strings to use.

In addition to the answer generated by the LLM, the query returns knowledge graph objects that model the documents, as well as additional objects such as instances of `Data_Product` and `prov:Activity` that are likely to be relevant to the user's prompt. This is done in two ways:

1. The SPARQL query returns the object that has the annotation or data property whose string produced a vector that meets or exceeds the minimum similarity score. One feature of a neurosymbolic knowledge graph is that it stores these links between vectors and knowledge graph objects automatically as part of the vector generation process.
2. Additional `OPTIONAL` patterns match general properties, such as `dp:is_downstream_of`, against the initial objects returned in step 1. This yields additional parts of the graph that can be visualized in a graph browser and are likely to be useful to the user. These additional nodes and links provide more entry points into the knowledge graph, making it easier to expand or contract nodes to explore related information. In this domain, one common example is expanding the graph to understand the lineage of data assets relevant to the current prompt, including the source assets, transformations, activities, and downstream dependencies associated with them.

In this proof of concept prototype, the user can invoke the Gruff graph browser on all of the knowledge graph objects that matched each query and further explore the Semantic Knowledge Graph. This enables the user to exploit the semantic models in the knowledge graph by further browsing the graph, e.g., by expanding nodes to show additional

linked objects. For details in other domains, see DeBellis et al. (2024) and DeBellis et al. (2025).

Olesen-Bagneux defines two fundamentally different types of data catalog search (Olesen-Bagneux, 2023):

1. *Searching **In Data***: "Searching in data is when we search in the actual data for something we want to know. For example, we might ask: 'How many people looked at our website last Saturday night?' The answer to this is a value from access records: 1,000 people looked at our website last Saturday night."
2. *Searching **For Data***: "Searching for data is when we search for the sources that contain the data we need. For example, we might ask: 'Where can we find data about traffic on our website?' The answer will be a data location."

In a Data Catalog built on a Semantic Knowledge Graph using a RAG, searching *in* data usually means searching the contents of text strings for documents and reports that aggregate data from tables, websites, etc. This is where RAG systems typically focus. However, searching *for* data (i.e., searching metadata) is at least as important for a data catalog.

In the Graph RAG tool, this information is primarily in the values of the knowledge graph, e.g., lineage data captured by PROV-O properties such as `prov:wasGeneratedBy` and `prov:wasDerivedFrom`. These axioms and values are typically not used to generate vectors. If not remedied, that would be a fatal barrier to searching *for* data. However, I have developed a simple set of Python functions (DeBellis, 2026) that generate what I call `embedding notes`. Embedding notes are strings generated via pattern matching and stored as values of the annotation property `dp:embedding_note`; they capture knowledge in the graph. These strings can then be assigned vectors. Using this technique, the RAG is able to understand how to search *for* data in addition to handling searches *in* data.

The examples following will, over the next three subsections, go from simple to sophisticated and will cover both searching in and

searching for cases. The question “Based on the latest advertising revenue report for 2025, which age segment is our primary revenue driver?” is an example of searching *in data*. The example that starts by asking “Are there any governance annotations in our data catalog relevant to the GDPR?” as well as the follow-up dialog are examples of searching *for data*.

Starting Simple: What Is This Catalog For?

I started simple with the most basic question I could think of: *“What is this catalog for?”*

The answer was correct, but the tone was off:

> *“The Data Catalog, such as the one used by StreamForge, is essentially a governance and management tool. It helps organize, define, and manage data resources like datasets, data models, and other components vital for the operation and decision-making within a company.”*

The content is correct, but it sounds like a Wikipedia article that happens to know about StreamForge. I corrected this by setting the **Complete LLM Prompt for chatState** parameter in the AllegroGraph LLM interface (Franz Inc., 2024) using a prompt provided to me by ChatGPT:

> *“You are a data catalog assistant for the StreamForge platform. StreamForge is a company that hosts user-created videos. Write answers in the voice of a StreamForge data team member speaking internally. Use first person plural (“we”, “our”). Avoid generic encyclopedia phrasing such as “X is…” or “such as the one used by StreamForge…”. Answer questions using information from the StreamForge Data Catalog and its associated documents. When possible, ground answers in specific Data Products, Documents, or metadata. Your users are employees of StreamForge, ranging from CXO executives and VPs to data analysts. {HUMAN} asks about {BOT} specific knowledge. Here is*

the appropriate knowledge relative to the question {EXPERTISE_HINTS}. Here are relevant previous questions from the user {HISTORY_HINTS}. Here is the transcript of the current conversation so far: {FEED}. Insert the list of citations whose hints informed the response into the 'citation_ids' array."[1]

The terms in all caps surrounded by curly brackets (e.g., {HUMAN}, {EXPERTISE_HINTS}) are template variables that act as placeholders within the prompt. At run time, the system replaces these variables with context-specific values, such as the user's question, retrieved knowledge graph content, prior conversation history, and citation metadata that points to the relevant knowledge graph objects for each response. This allows a single prompt template to be reused while incorporating dynamic, query-specific information into each request.

This made all the difference. Now when I asked, *"What is this catalog for?"* I received a much better answer:

"Our StreamForge Data Catalog is designed to effectively manage, discover, and utilize our data products and metadata. It ensures we have access to detailed descriptions, documents, and usage guidelines for each data asset, aiding in enhancing data-driven decisions and supporting new feature creation."

Next, I tried another simple question: *"What is a Bounded Context?"*

"Each business domain of our catalog is defined by its own Bounded Context, allowing us to have distinct boundaries for different domains. This means that each domain has its own models, vocabulary, and rules as well as maps that explicitly model dependencies between polysemes. This maintains clarity, avoids data silos, and minimizes the time that must be spent when communicating and integrating between different systems and teams."

Next, I tried a business-focused question: *"Based on the latest advertising revenue report for 2025, which age segment is our primary revenue driver?"*

The question, answer, and matching document are shown in Figure 13-12. This is a simple user interface I've developed for various RAG systems using the Python library streamlit.io (Snowflake Inc., 2024). The numbers at the top are parameters that control the text strings that will be used to generate answers. The **Relevance Threshold** sets the minimum similarity score for a retrieved text string to be considered relevant to the prompt. Similarity is computed using cosine similarity (Anastasiu & Karypis, 2016). A higher threshold requires a closer match. The **Number of Corpus Documents to Use** determines how many matching text strings are supplied to the LLM.

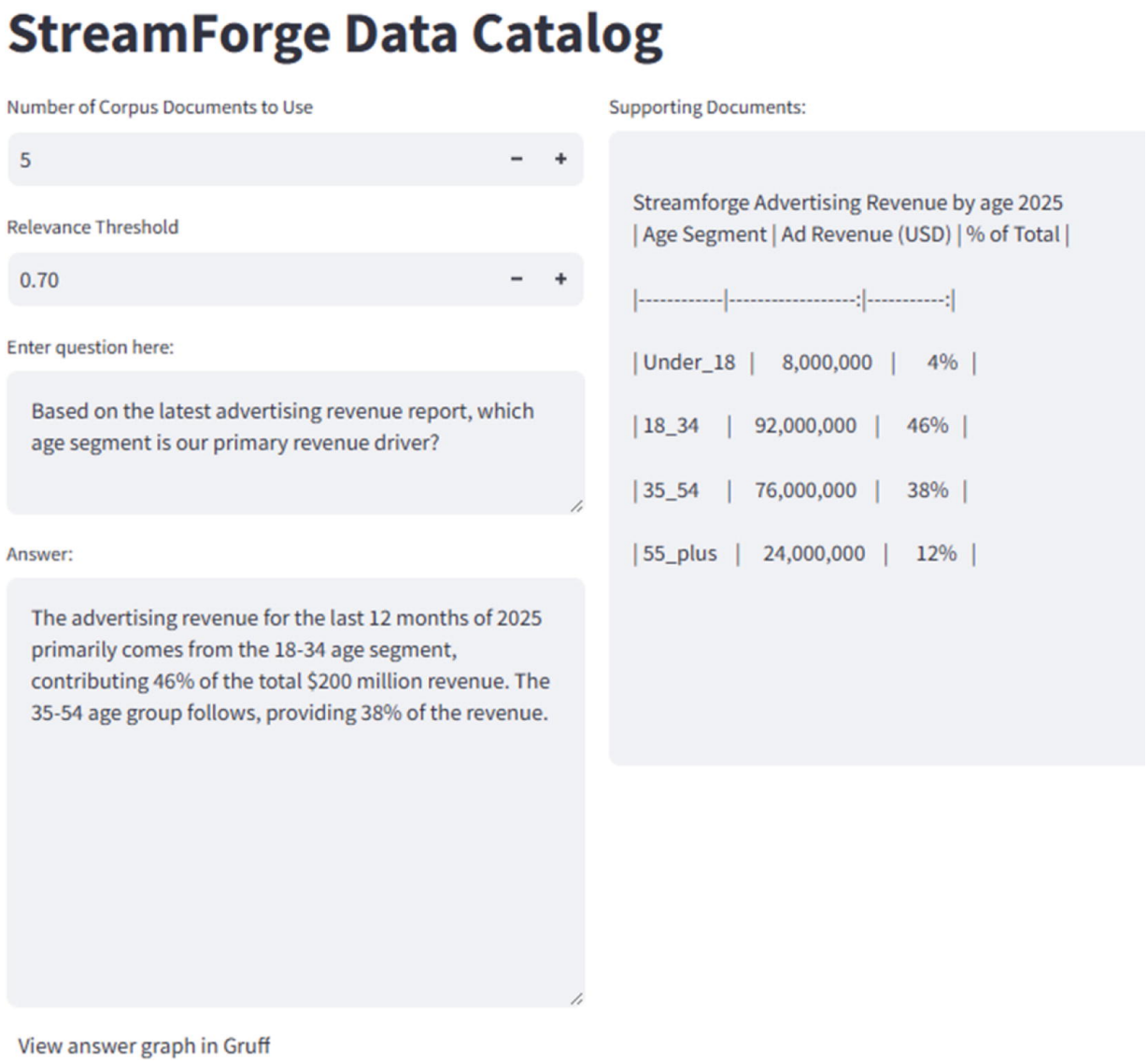


Figure 13-12 The Data Catalog RAG basic user interface

Combining LLM and Graph Results: Governance Annotation Example

This section demonstrates that Graph RAG is not limited to simple "ask a question, get text back" search.

Instead, a single SPARQL query integrated with an LLM can

1. Bind a specific slice of the knowledge graph (e.g., a data product, its generating pipeline, and its source datasets)
2. Assemble those facts into a controlled grounding string

3. Pass that grounding into an LLM call

4. Return the LLM's natural-language answer plus the underlying graph nodes used to construct it

From the user's perspective, this feels like a single interaction. Under the hood, it is a deliberate fusion of symbolic structure and statistical language modeling. To further illustrate the power of combining a Semantic Knowledge Graph with a large language model, consider the following scenario: StreamForge is branching out from serving only the United States to Europe and hence must conform to the European Union GDPR laws. This allows a workflow such as

- Show any data products that have ODRL governance annotations related to GDPR.
- Describe the resources required to address the annotation.
- Describe which data products are downstream of the data product with the GDPR issue. This gives us the relevant assets that will be impacted if the data product is changed or taken offline.
- Display the relevant entities and properties in the knowledge graph using a graphical browser such as Gruff and allow the user to further browse the knowledge graph.

We start with the following natural-language dialog with the StreamForge Data Catalog:

Prompt: *Are there any governance annotations in our data catalog relevant to the GDPR?*

Response: *There is a Governance Annotation specifically relevant to the GDPR called the 'GDPR Risk Flag for Customer Profile Data Product.' This annotation is an instance of Risk Flag Annotation and has several attributes that make it crucial for GDPR compliance:*
***Status**: Active*
***Downstream Activity**: It is downstream of the Create Annotation 1001 Activity.*

***Review Requirement**: Requires review by the Security Team.*
***Attribution**: Attributed to Security Engineer 1.*
***Generation Activity**: Generated by Create Annotation 1001 Activity. It is defined as: "This data product may contain fields that qualify as personally identifiable information (PII) under GDPR. A formal privacy review is required before expanding access beyond the analytics team."*

Prompt: *Which data products is the GDPR risk flag attached to?*

Response: *The GDPR Risk Flag is attached to the 'Customer Profile Data Product'.*

Prompt: *Which data products are downstream of Customer Profile Data Product?*

Response: *The Customer Profile Data Product serves as an upstream data source for:*

1. ***Recommendation Data Product:** This data product is downstream of Customer Profile Data Product and uses it.*
2. ***SF Creator Insights Data Product:** This data product also utilizes Customer Profile Data Product as one of its sources.*

Prompt: *Who needs to review the GDPR Risk Flag?*

Response: *The GDPR Risk Flag for the Customer Profile Data Product requires review by the Security Team.*

Prompt: *Are there any members of the security team who have GDPR as a skill and whose work ratio is less than 50%?*

Response: *Security Engineer 2 is a member of the security team who holds the GDPR Skill and has a workload ratio of less than 50%.*

At this point, stop and think for a minute about how much work it would be to answer the above questions in a conventional systems development effort. Then, imagine that effort continuing, growing, changing, and experiencing personnel turnover for, say, ten years. Stop and think again about how much work this would now entail, as well as

how great is the likelihood that the information needed would even have been preserved.

Figure 13-13 shows the knowledge graph objects returned with the last response.

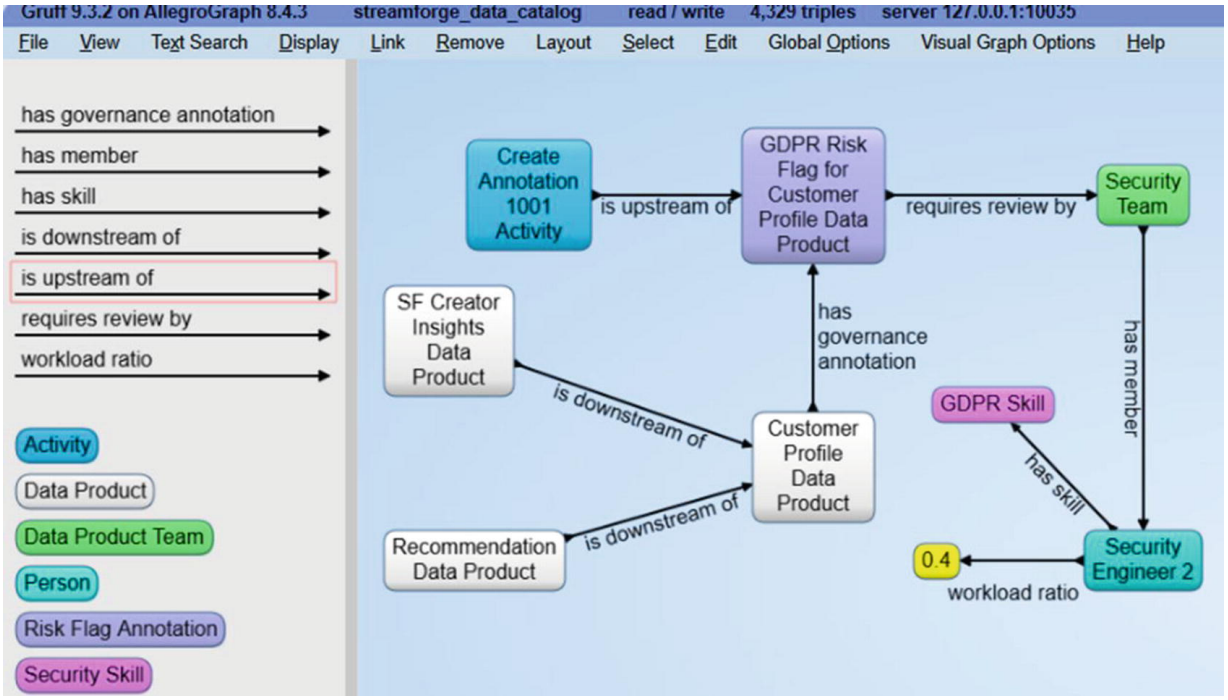


Figure 13-13 Knowledge graph returned with response from GDPR dialog

Figure 13-14 shows an example of how the user can utilize the graphical browser to focus on a specific part of the graph. In this case, starting with the graph from Figure 13-13, the user focuses on expanding information about the lineage of the data products that use the data product with the GDPR flag.

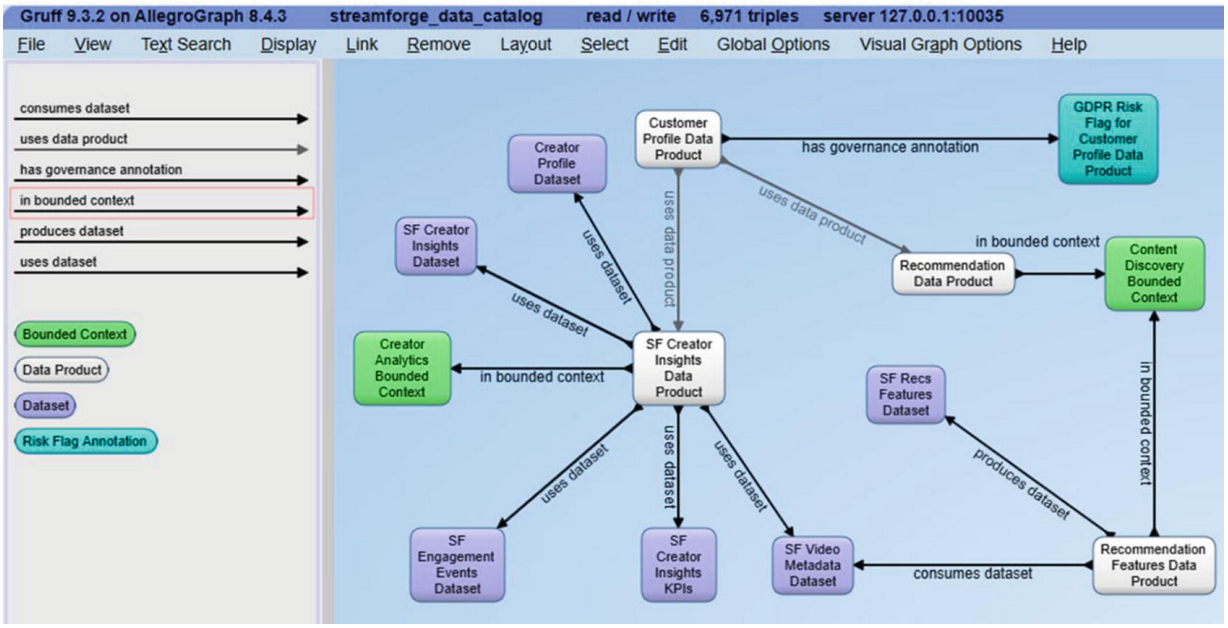


Figure 13-14 User redraws graph from Figure 13-13 to focus on lineage of relevant data products

Provenance and Lineage As First-Class Citizens

These examples illustrate the value of modeling data products and a data catalog as a Semantic Knowledge Graph.

In the StreamForge catalog, the PROV-O standard ontology models data provenance:

- Metrics and Data Products are modeled as `prov:Entity`.
- Pipelines and runs as `prov:Activity`.
- Teams and systems as `prov:Agent`.
- Relationships such as `prov:wasGeneratedBy`, `prov:used`, and `prov:wasDerivedFrom` explicitly encode lineage.
- New super-properties defined in the `data_product` ontology such as `is_upstream_of` and `is_downstream_of` provide practical abstractions over more specific lineage relationships, allowing generic queries to retrieve related catalog assets even when the underlying links are expressed with different PROV-O or domain-specific properties.

The ODRL standard ontology models policies, governance annotations, permissions, and prohibitions.

Concepts such as data products, team ownership of products, skills, and workloads are modeled by new classes and properties defined in the data product ontology such as `Data_Product`, `Data_Product_Team`, `has_skill`, `workload_ratio`, `is_downstream_of`, and `is_owned_by`.

When this structure is included in Graph RAG queries, the system can answer questions that are traditionally very difficult for LLM-only systems, such as

- Where did this executive dashboard number come from?
- Which datasets and pipelines were involved?
- Who is accountable if we have to pause this data product?
- Who has the appropriate skills and bandwidth to deal with specific issues?

Why Textual Annotations Still Matter

One practical insight from this work is that today's LLMs are still far better at consuming text than pure graph structure. As a result, it can be useful to augment structured relationships with explanatory text, for example:

- Including lineage summaries in `skos:definition`.
- Auto-generating natural-language descriptions from graph relationships that define subclasses, superclasses, and instances. This is done in this example by the Python script: https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/data_product_src/add_embedding_notes_from_classes.py.
- Auto-generating natural-language descriptions from property values. This is done in https://github.com/mdebellis/SemanticKG-Design/blob/main/data_product/data_product_src/add_embedding_notes_from_properties.py.

These files transform the knowledge in the graph into textual annotations stored using `dp:embedding_note` (DeBellis, 2026). We include this property when we generate vectors for the knowledge graph, and hence, all the relevant knowledge from the graph is captured in a form that the LLM can understand and utilize.

This is a reflection of current tooling. It is reasonable to expect that

1. Vendors will continue improving native graph-aware LLM integration
2. Developers will increasingly auto-generate textual annotations from structured models

In fact, in a strong illustration of the pace of innovation with this technology, a new tool from AllegroGraph called GraphTalker (Franz Inc., 2026) that provides features similar to Graph RAG was released while I was in the middle of writing this chapter. It will be discussed below.

More Than One Path Through the Graph

The integration of retrieval-augmented generation (RAG) with enterprise metadata can transform a Data Catalog into a conversational interface. Natural language questions, semantic similarity, and structured visualization create a powerful discovery mechanism. The RAG Bot is more than a search tool; it is a collaborator and can help the user formulate solutions.

A long-standing principle in user interface design is that systems should provide multiple ways to accomplish the same task. As described in usability research, e.g., Nielsen's work on "flexibility and efficiency of use" (Nielsen, 2024), different users have fundamentally different working styles. Some prefer precise, structured queries, while others prefer more exploratory, conversational interaction. Also, different problems are better suited to different user interfaces.

This principle applies directly to search. Even in systems that incorporate large language models, keyword search remains essential. Keyword search supports precision, transparency, and rapid iteration, allowing users to experiment safely and refine their understanding through immediate feedback. LLM-based search, by contrast, excels at handling ambiguity and synthesizing information but is less predictable and less transparent. Keyword search is also often preferable for power users who know the data catalog and data product names very well and prefer simple, fast access.

Providing both approaches is not redundancy. It is necessary to support different cognitive styles, support different types of questions, and to create an interface that invites exploration. Engineers may wish to locate all resources containing a specific identifier. Governance teams may need to search for policy references. Analysts may simply want to find every catalog entry that mentions a particular term.

One of the practical advantages of using a Semantic Knowledge Graph is that these capabilities coexist naturally. In addition to logical inference (OWL), constraint validation (SHACL), and conversational retrieval (RAG), most enterprise graph platforms also provide free-text indexing. This allows keyword search to be layered directly onto structured metadata with minimal additional modeling.

Creating a Free-Text Index (FTI) (Wilkins, 2023) over selected properties of the StreamForge Data Catalog requires only a small amount of configuration of tools in AllegroGraph. Once defined, the index can be queried using traditional keyword search. The FTI can also be included in SPARQL queries, enabling precise keyword search alongside semantic reasoning and vector-based retrieval.

This illustrates a broader architectural theme of this book: a Semantic Knowledge Graph does not force a single access paradigm. Instead, it supports *multiple complementary modes of interaction—logical querying, keyword search, and semantic similarity—within a unified framework.*

The implementation of the FTI can be found at https://github.com/mdebellis/SemanticKG-

Design/blob/main/data_product/data_product_src/sf_fti.py.

To create an FTI, all that is required is to

1. Define the data and annotation properties to utilize. Typically, as with vector embedding, only properties with a range of string or literal are used.
2. Define the "stemming" function. This is a function that reduces related word forms to a common stem.
3. Define the "stop words." These are words that will be ignored when creating the FTI such as "and," "or," and "the."

I defined two FTIs. One called `sf_name_search` that only uses label properties: `rdfs:label`, `skos:altLabel`, and `skos:prefLabel`. This is the more conservative search tool when you are looking for entities by name. The other is `sf_content_search`. This uses all the properties for the name search as well as most of the properties used for vector embedding. This is the tool that casts a wider net that you use when you are looking for a document that contains a specific phrase.

Figure 13-15 is an example of using this type of search. The user picks which FTI to use and enters an expression. They are looking for all entities that have "Video" and "Data Product" in their labels or content.

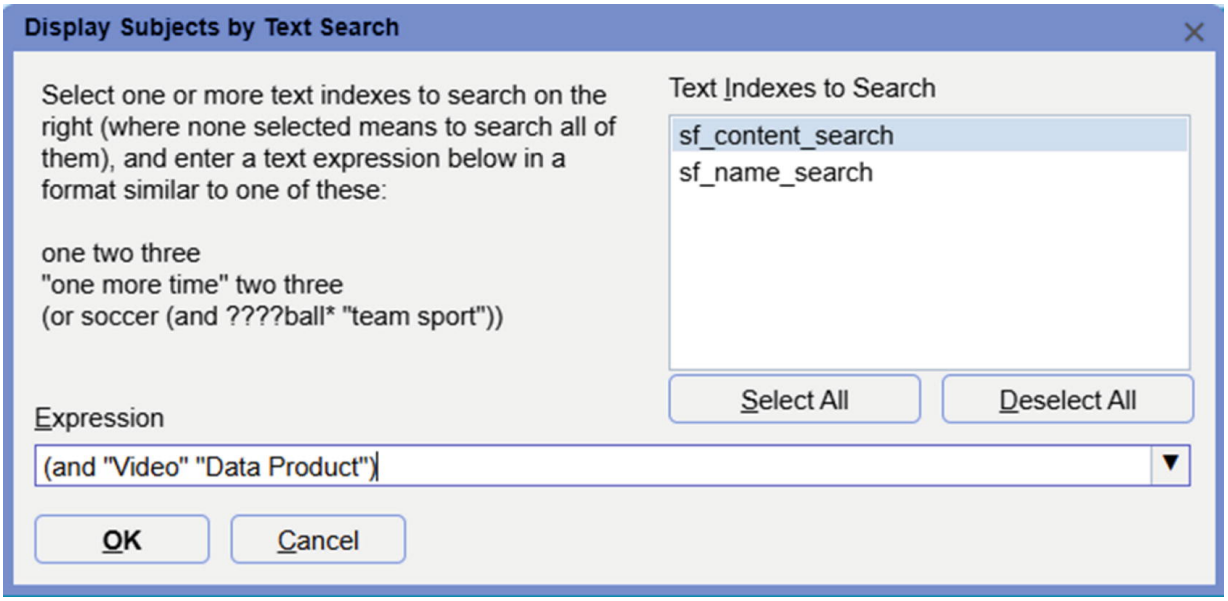


Figure 13-15 Using keyword search

Figure 13-16 shows the results of this search with some additional nodes and links added by the user. The results of the search were the port, data products, and document chunks at the bottom of the graph. I added the type of each node (the class it is an instance of) and the class that is the superclass for those classes. Note that for the instances of `Document_Chunk`, the text strings are not in the label but in the content of one of the properties.

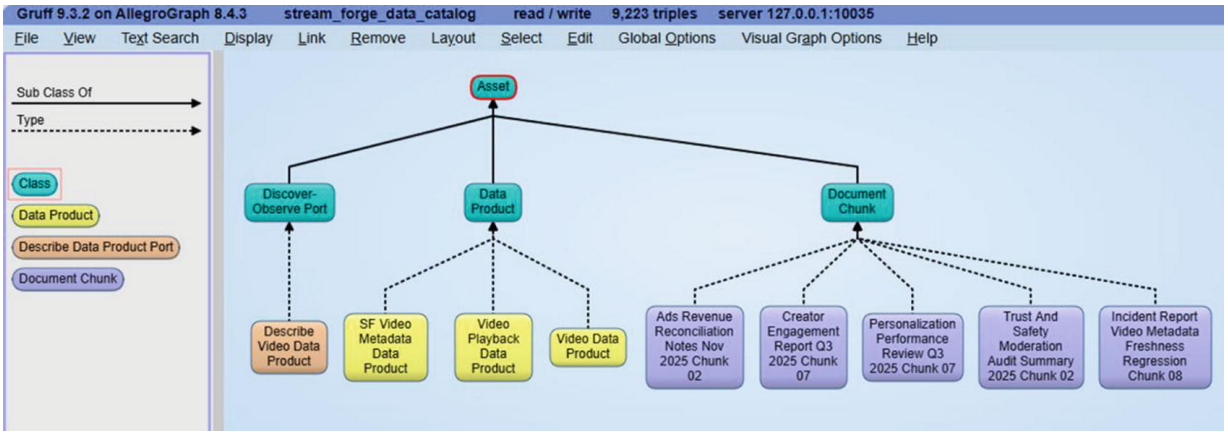


Figure 13-16 Results of search in Figure 13-15 with addition of type and subclass links

GraphTalker

“Well, in our country,” said Alice, still panting a little, “you’d generally get to somewhere else—if you ran very fast for a long time, as we’ve been doing.”

“A slow sort of country!” said the Queen. “Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!”

—Through the Looking-Glass, and What Alice Found There. Lewis Carroll.

Since LLMs turned the tech world upside down, I’ve often thought of this quote because that’s how it feels to work in information technology sometimes. Things are changing so fast, we have to run as fast as we can, just to stay in place. The section above was originally the end of this chapter. Then, I finally had time for something that my colleagues at Franz Inc. had been encouraging me to look at for a while. A new tool called GraphTalker (Franz Inc., 2026).

As I experimented with GraphTalker, I realized I had another section to write. GraphTalker takes a very different approach than a Graph RAG chatbot. A RAG chatbot uses the corpus (and strings generated from it) as well as the history of the chat to respond to prompts using an LLM.

In contrast, GraphTalker provides a conversational interface to an AllegroGraph knowledge graph. Instead of requiring the user to write SPARQL directly, it accepts a natural-language question and uses an LLM (Claude from Anthropic) to help construct SPARQL queries over the selected repository. GraphTalker uses the graph itself as part of the query process: it inspects the repository schema, uses SHACL-based descriptions where available, consults examples in the **Query Library**, executes generated SPARQL queries, and iteratively revises them based on the results or errors returned by AllegroGraph. The final answer is then synthesized from the actual query results and can include explanatory text, tabular data, and visualizations. GraphTalker is an

agentic natural-language layer over SPARQL. Figures 13-17 and 13-18 show the dialog regarding GDPR annotations in GraphTalker.

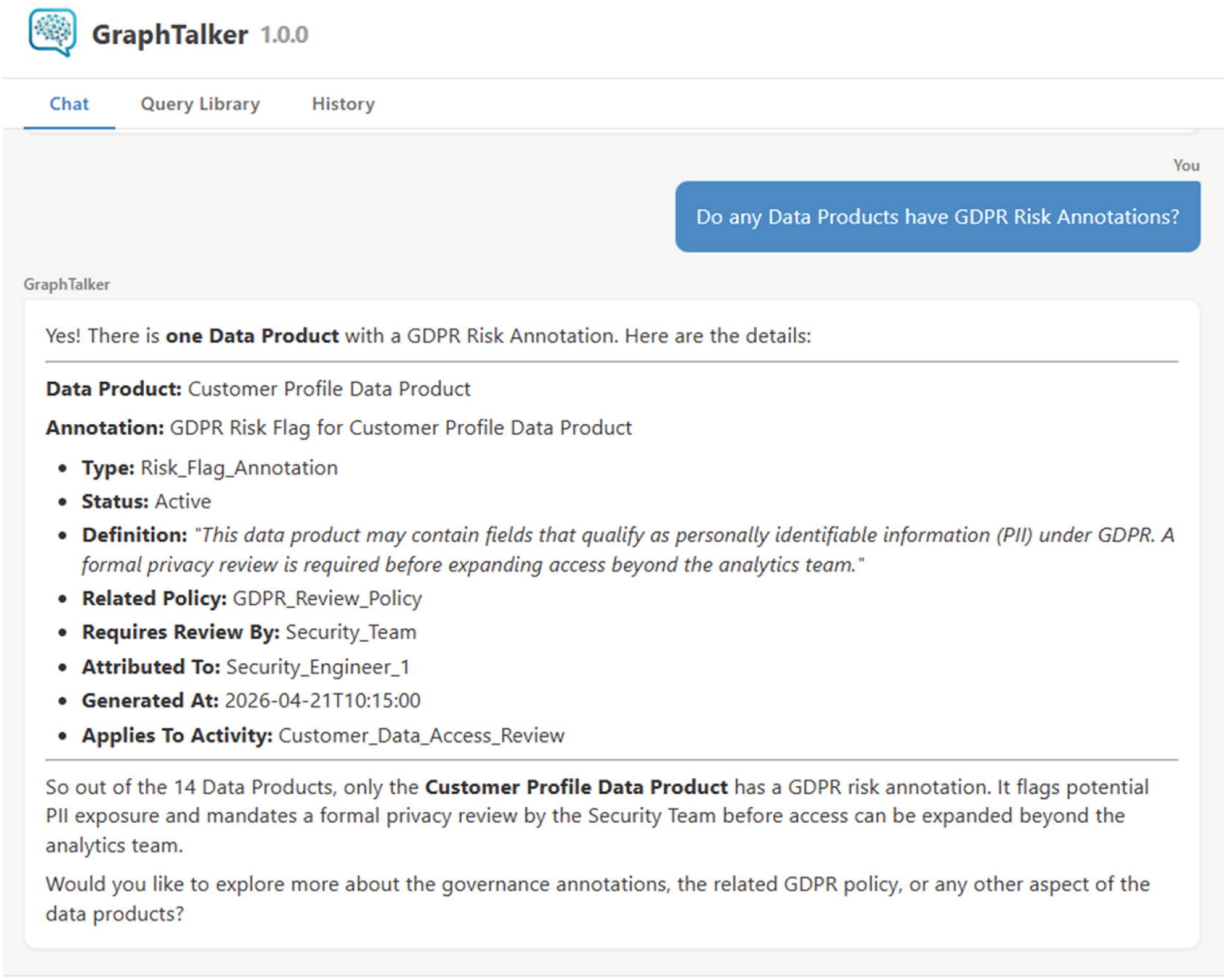


Figure 13-17 Querying the Data Catalog about GDPR Annotations with GraphTalker

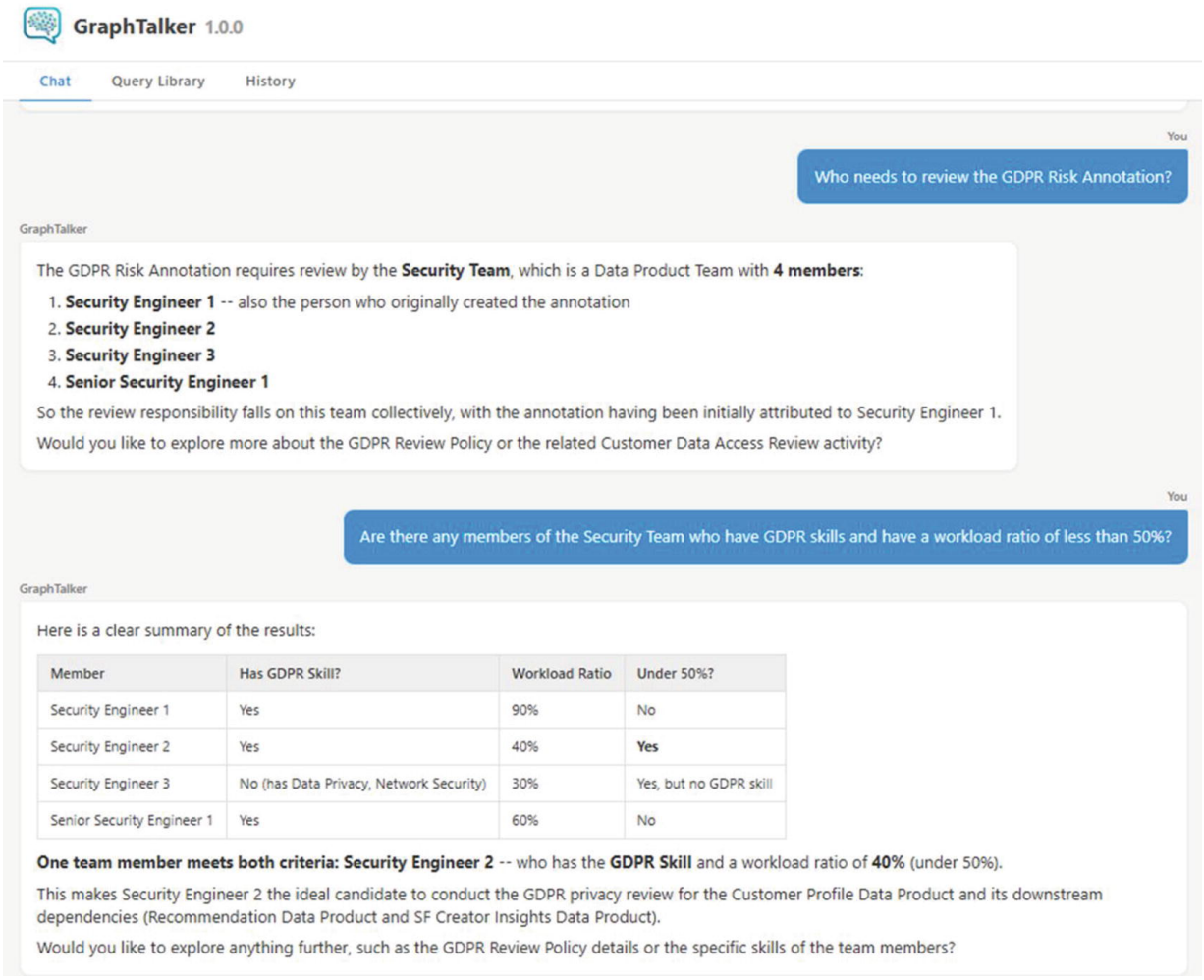


Figure 13-18 Querying the Data Catalog about resources to address the GDPR issue with GraphTalker

The GraphTalker examples above show the strengths of an agentic graph interface. When asked which data products have GDPR risk annotations, who needs to review those annotations, which team members have the right skills and availability, and which downstream data products might be affected, GraphTalker was able to use the structure of the Semantic Knowledge Graph directly. These are not simply text retrieval questions. They require following relationships among data products, governance annotations, policies, teams, people, skills, workload ratios, and lineage dependencies. In these cases, GraphTalker behaves less like a document search engine and more like an interactive analyst that can inspect the graph, generate SPARQL

queries, evaluate results, and translate those results into a readable answer.

GraphTalker understands the knowledge in the graph in a deeper way than the simple annotation note solution that I developed. In addition, it doesn't require the generation of vectors. It also provides formatting and a usable "out of the box" user interface. In my initial experiments, GraphTalker seemed to have an important limitation. When asked, *"Based on the latest advertising revenue report for 2025, which age segment is our primary revenue driver?",* GraphTalker was not able to produce a useful answer. The answer was available in a document chunk and was easy for the RAG system to find. However, as I learned more about GraphTalker, I realized all I needed to do was to add an extra sentence to the prompt. My new prompt was:

> *"For the next prompt, if you cannot find a result in the structured data, then investigate if the answer might be found in the text via a RAG query. Based on the latest advertising revenue report for 2025, which age segment is our primary revenue driver?"*

With this prompt, GraphTalker found the answer as shown in Figure 13-19.

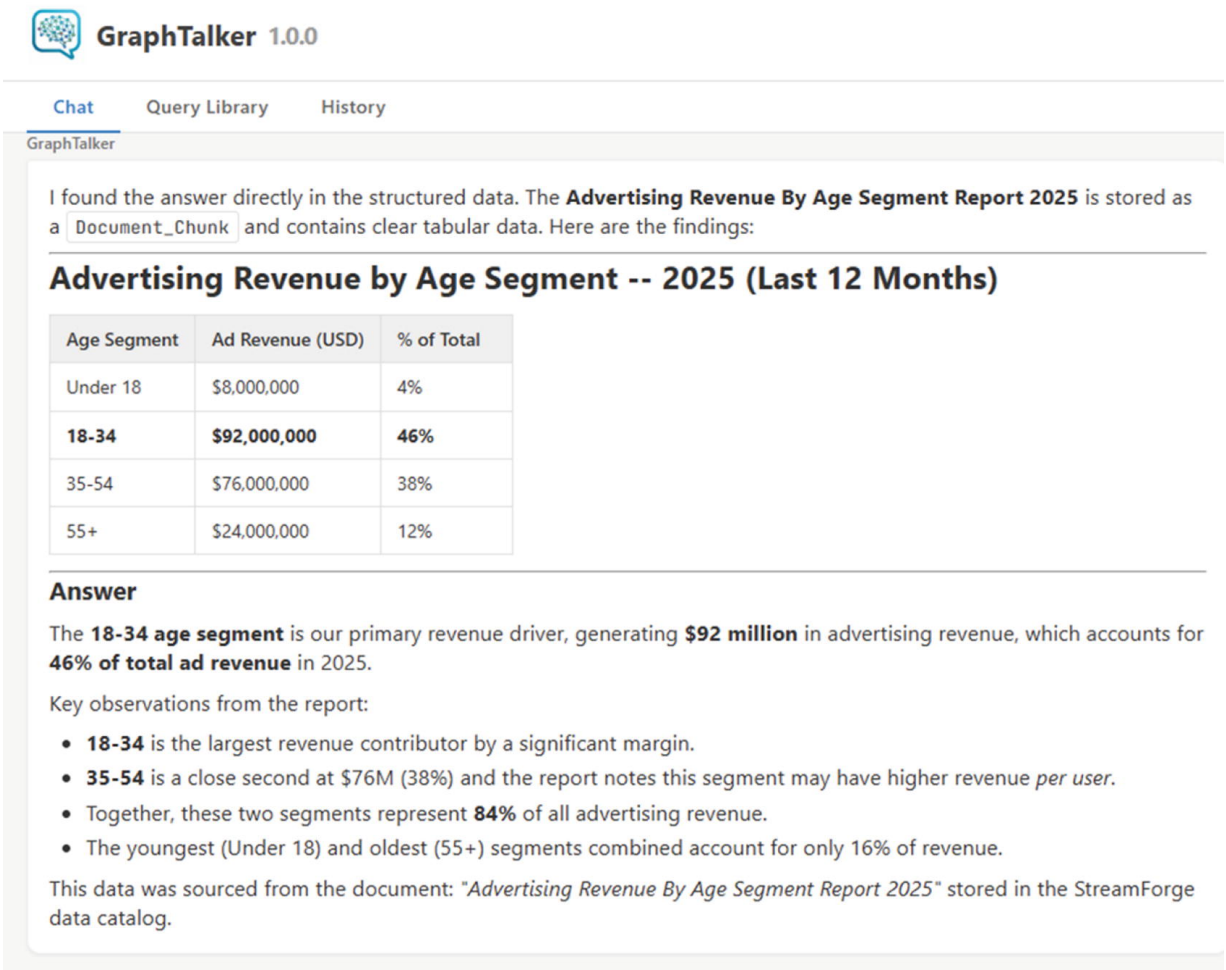


Figure 13-19 Adding a RAG sentence to the prompt allows GraphTalker to search **in** as well as **for** data

In addition, GraphTalker provides the capability to define tutorials (Franz Inc., 2026) that add additional information like the extra sentence used in the above example to all prompts automatically.

Benefits for Decision-Makers

The examples utilizing Graph RAG, FTIs, and GraphTalker illustrate

- A realistic enterprise pattern inspired by work I did on a data catalog for a large Internet company.
- Capabilities that leverage natural language, LLMs, graph visualization, agents, and keyword search in an integrated platform.

- The advantage Semantic Knowledge Graphs provide via standard reusable ontologies that allow developers to utilize best practices for well-understood problems so that they can focus on their special services and products that provide them with competitive advantage.
- The power of having developers build with Semantic Knowledge Graphs and LLMs. All the example knowledge graphs, Python code, SHACL, and other artifacts that can be found in the book's GitHub repository were developed by me as I was writing this book and required minimal effort.

The Data Catalog RAG and GraphTalker illustrate a fundamental concept I've often observed in my recent work: LLMs and Semantic Knowledge Graphs (SKGs) are powerful precisely because their strengths are complementary. LLMs provide flexibility, natural language fluency, and the ability to work with incomplete, informal, or ambiguous information. SKGs provide explicit semantics, formal structure, reusable domain knowledge, and explainable relationships. Together, the LLM makes the graph easier to query and apply, while the graph grounds the LLM in a more reliable and transparent model of the domain.

For executives and senior leaders, the significance is straightforward:

- Answers are available immediately and can be found with virtually no training.
- Additional or specific technical detail is available for users who require it via graphical visualization and other specialized tools.
- We can apply the power of LLMs to analyze enterprise data without the issues of hallucinations and black-box reasoning.
- Explanations are traceable.
- Assumptions can be inspected.
- Governance and lineage are not hidden in neural networks but explicitly represented and visualized in logic and graphs.
- Enterprise data modeled in a Semantic Knowledge Graph provides an incredibly powerful platform for corporations to develop new types

of unprecedented systems.

This combination of natural language plus explicit structure makes the combination especially compelling in enterprise settings.

The examples here used a small Semantic Knowledge Graph data catalog as an example. However, the example applies to all enterprise data. Enterprise knowledge management or a data lake could just as easily be used as an example as shown by the case studies in the next chapter. Indeed, as LLMs and Semantic Knowledge Graphs are adopted by industry, the boundaries between these types of systems will become more and more ill-defined. A Data Lake also requires a Data Catalog to provide input and control over data to ensure that organizations abide by rules such as HIPAA and GDPR. A knowledge management system requires analysis of historical data in the data lake to provide executives and knowledge workers with timely information that leads to informed strategic decisions. The needs are essentially the same regardless of the application and architecture, and the combination of Semantic Knowledge Graphs and large language models works equally well on all types of enterprise data.

Even more compelling, all this is just the beginning. New types of tools such as GraphTalker are adding even more capabilities to leverage enterprise data. The organizations that will succeed in the future will be those who can leverage these new technologies and use them to solve domain problems in unprecedented ways.

Summary

- *Domain-Driven Design (DDD) is the most widely used approach to modeling microservices and Data Products*. One of the core concepts of DDD is the bounded context. Bounded contexts define encapsulation boundaries. However, they do so not only at the level of software components but also at a conceptual level based on the semantics of domains. This makes OWL and SHACL particularly well suited to modeling DDD concepts.

- *The data product is a very important new concept that seeks to do for data what Microservices have done for domain services*: encapsulate responsibility, define clear contracts, and enable rapid, agile change.
- *Semantic Knowledge Graphs are an excellent platform for modeling data products and Data Catalogs*. The W3C vocabularies PROV-O, ODRL, and DCAT provide a very solid foundation for such development.
- *Connecting an LLM to a knowledge graph data catalog using a retrieval-augmented generation architecture provides a powerful tool that leverages the strengths of both technologies*. Semantic Knowledge Graphs provide explicit knowledge representation, semantics, and graph query. LLMs provide the ability to not just search the catalog but ask questions requiring reasoning.
- *Even with an LLM, conventional search still has value*. Such search is trivial to implement with Semantic Knowledge Graph platforms that support NLP tools such as a Free Text Index.
- *GraphTalker illustrates a different architecture for combining LLMs with Semantic Knowledge Graphs*. Rather than simply retrieving text or graph-derived snippets for the LLM, GraphTalker uses the LLM as an agentic interface to the graph itself: it can inspect the schema, generate and execute SPARQL queries, refine those queries when needed, and synthesize answers from the results. This demonstrates how rapidly the field is moving from conversational search toward interactive reasoning over formal semantic models.
- *Modeling using Very High Level Languages such as OWL and SHACL takes some extra effort, but that effort pays huge dividends* by enabling various types of search and interaction and by making implicit assumptions from models and documents explicit knowledge representation.
- *Further Reading:* For more on using Semantic Knowledge Graphs for retrieval-augmented generation (RAG): Stardog white paper on Graph RAG (Clark, 2024). For data catalogs: Enterprise Data Catalog (Olesen-Bagneux, 2023). For more on Domain-Driven Design:

Domain-Driven Design (Evans, 2003). For more on GraphTalker, see (Franz Inc. 2026a).

Footnotes

1 The need for clarifications like this illustrates how LLM outputs are only as good as the user's input prompts, and why architectures such as RAG that combine LLMs with explicit knowledge representation are so important for many mission-critical and enterprise applications.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_14

14. Semantic Technologies: Industry Applications

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

This book has shown how Semantic Knowledge Graphs provide technology that plays an important role in modern enterprise architectures. The technology provides key features that are required for Data Fabric and Data Mesh architectures such as active metadata, semantic metadata, graph modeling, and querying. In addition, the explicit knowledge representation, explanations, and logical reasoning are the perfect complement to the statistical but black-box reasoning of large language models (LLMs). The following case studies illustrate these points in diverse industry domains working with big data.

Case Study 1: Microsoft Docs—Utilizing Graphwise to Define Semantic Structure at Enterprise Scale

The challenge: Content at enterprise scale

Microsoft maintains one of the world's largest and most complex technical documentation ecosystems. This includes 450 GitHub repositories, 3.5 M pieces of content, 4,500+ authors and documentation for 45 Microsoft products.

Microsoft Docs spans millions of documents covering products, APIs, frameworks, and services that evolve continuously. Content is authored and maintained by globally distributed teams, often using different terminologies, styles, and assumptions about audience and usage. Their existing infrastructure for document management was fragmented, unstructured, and had no unified content model. This made it increasingly difficult for Microsoft to manage their documents and for employees and customers to find the appropriate documents when they were needed.

As the documentation corpus grew, Microsoft faced a familiar but increasingly severe set of problems:

- Search results were inconsistent and often incomplete.
- Related content was difficult to discover or reuse.
- Terminology drift caused confusion across teams and products.
- Manual classification and tagging did not scale.

These challenges were operational failures caused by the scale and heterogeneous nature of all the documents required by Microsoft. Traditional keyword-based approaches and folder hierarchies were no longer sufficient to manage meaning across such a large and dynamic knowledge base.

The Semantic Approach: Making Meaning Explicit

To address these issues, Microsoft adopted a semantic approach to content governance using Graphwise Semantic Knowledge Graph technology (Graphwise, 2025a). Rather than treating documents as isolated text artifacts, the system introduced an explicit semantic layer that captured the meaning of content independently of its textual representation.

Key elements of this approach included

- Controlled vocabularies and concept models shared across teams
- Automated semantic tagging of documents based on meaning, not just keywords
- Explicit relationships between concepts, products, and topics

- Separation of content management from knowledge organization

This semantic layer enabled consistent interpretation of content across products and teams, even as the underlying documents continued to evolve rapidly.

Importantly, this was not framed internally as a “taxonomy project” or an “ontology initiative.” It was a response to concrete operational needs: search quality, reuse, governance, and scalability.

Outcomes and Impact

By introducing semantic structure, Microsoft achieved measurable improvements:

- Turned 3.5 M content pieces from 450+ repositories into a unified, intelligent documentation platform
- Automated semantic classification and discovery with >70% initial accuracy
- Dramatically improved search relevance and discoverability
- Reduced manual effort for content classification and maintenance
- Cut taxonomy management from weeks to hours and automated eight months of manual tagging effort
- Delivered smarter internal search and recommendations, ensuring scalable knowledge governance
- Greater consistency in terminology across products and teams

The result was going from a chaotic collection of documents to a scalable foundation for content governance as the corpus continued to grow.

The key lesson from this case is that one of the main drivers for Semantic Knowledge Graphs is to manage documents with technology that explicitly models versioning, dependencies, and most importantly the semantics of the documents. At this scale, meaning *must* be modeled explicitly or the documents become unmanageable.

Why This Matters

This case illustrates a recurring pattern in enterprise systems: semiformal models such as controlled vocabularies and taxonomies arise naturally as pragmatic solutions to real problems. Once in place, they serve as a foundation for more formal ontologies, even if that transition is gradual and incremental.

The Microsoft example demonstrates that semantic structure is not an optional enhancement—it is a necessary architectural response to scale.

Case Study 2: A Semantic Layer for an Automotive Enterprise Data Lake

The challenge: Making enterprise Data Lakes findable, governed, and reusable

Large industrial organizations increasingly rely on data lake architectures to store and process massive volumes of heterogeneous data. In the automotive domain, this includes sensor streams, simulation outputs, test results, and derived analytic datasets. While data lakes excel at scalable storage, they often fail at higher-level concerns such as data discovery, provenance tracking, access control, usability, and reuse. This case study describes how Robert Bosch GmbH addressed these challenges by introducing a semantic layer built on open standards and integrated with a modern, event-driven data architecture (Dibowski et al., 2020; Dibowski & Schmid, 2021).

Problem Context

Robert Bosch GmbH designs and manufactures automotive components. These components range from safety products such as airbags and electronic stability control to next-generation automated driving systems. The volume and type of data generated by these systems have been increasing rapidly in both volume and variety. The types of data range from sensor data to textual data and metadata

produced by the various projects as they research, design, manufacture, and improve their products.

Bosch operates a large-scale automotive data lake used to support advanced driver-assistance systems and autonomous driving research. The data lake aggregates data from a wide range of sources, including vehicle test fleets, simulation environments, and external databases. Although the data lake provided centralized storage, users nevertheless found themselves struggling to answer some very critical and fundamental questions:

- What data exists, and how can it be found?
- Where did a dataset come from, and how was it derived?
- Who is allowed to access or modify specific data assets?

These are challenges that virtually all enterprise data lakes face. The initial appeal of building data lakes using open-source software such as Hadoop is well illustrated by this example. The data lake is far less complex and more affordable than the cube model of a data warehouse. However, over time, the unstructured models used by data lakes result in "Data lakes [becoming] data swamps" (Menon, 2022). This is an ideal use case for Semantic Knowledge Graphs.

Architectural Approach

Bosch addressed these issues by introducing a Semantic Data Lake Catalog as a layer on top of their existing data lake. Rather than replacing existing storage or processing systems, the semantic layer acts as a unifying metadata and knowledge management component.

At the core of the solution is an OWL-based ontology called DCPAC (Data Catalog, Provenance, and Access Control), which integrates several well-established W3C standard vocabularies such as (see Figure 14-1)

- DCAT for dataset and catalog metadata
- PROV-O for provenance and lineage
- ODRL for access control policies
- SKOS for semantic tagging and controlled vocabularies

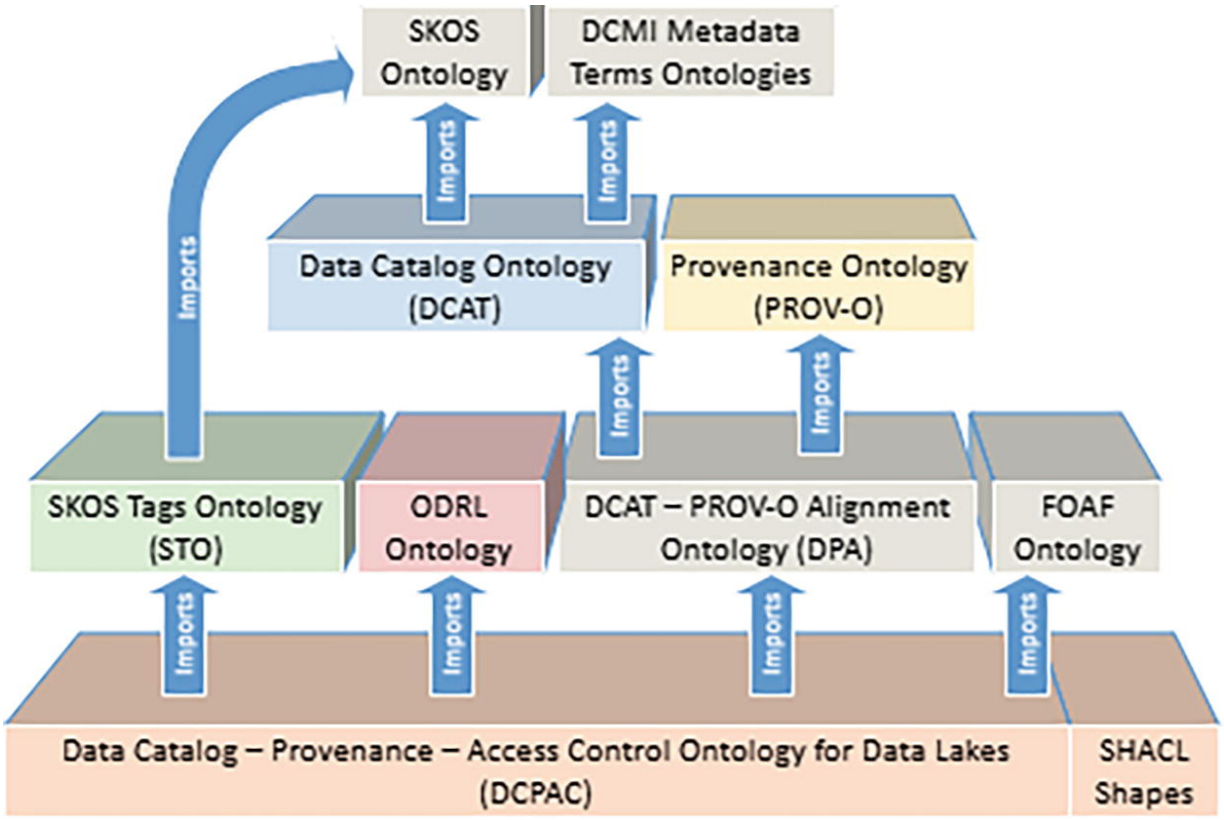


Figure 14-1 Vocabularies used for a semantic data lake for an automotive components manufacturer. Figure reproduced with kind permission of Robert Bosch GmbH and the project team

This combination allows datasets, services, and distributions in the data lake to be described in a uniform, machine-interpretable way, while remaining aligned with existing web standards.

Figure 14-2 illustrates the high-level architecture of the Semantic Data Lake Catalog and its role within the broader automotive data lake. The diagram shows how ingestion and processing pipelines publish events, how those events drive semantic catalog population, and how the resulting knowledge graph serves as a central point for discovery, governance, and access across heterogeneous data stores.

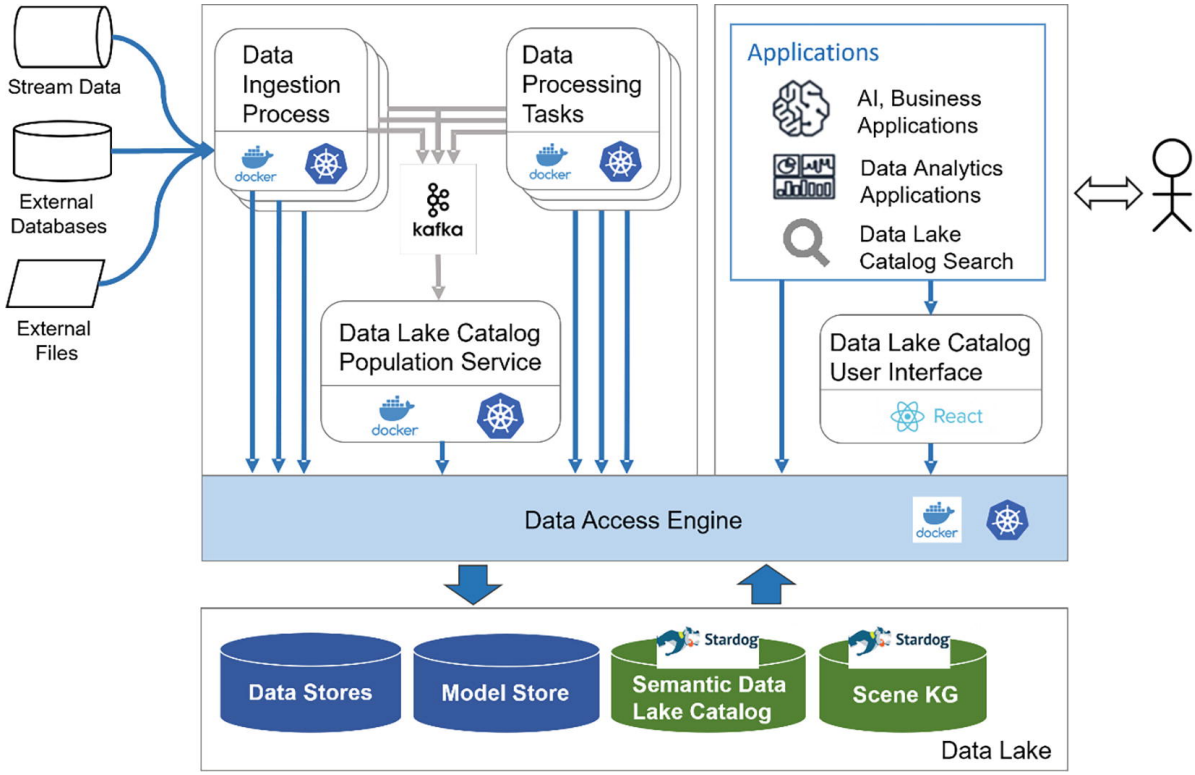


Figure 14-2 Architecture of the semantic data lake catalog, showing integration with data ingestion pipelines, event streaming, semantic metadata population, and governed access to heterogeneous data lake resources. Figure reproduced with kind permission of Robert Bosch GmbH and the project team.

Event-Driven Metadata Population

A key strength of this architecture is that semantic metadata is populated automatically, not manually. Data ingestion and processing pipelines publish standardized events to Apache Kafka whenever new data assets are created or modified. These events trigger a catalog population service that

1. Extracts technical and business metadata
2. Maps schema elements and tags to SKOS concepts
3. Records provenance information using PROV
4. Applies access control policies using ODRL

This event-driven approach ensures that the semantic catalog stays synchronized with the evolving contents of the data lake, even at very large scale.

Domain Semantics and Semantic Search

To support domain-specific discovery, Bosch developed an Automotive Signal Ontology that models automotive signals, sensors, components, and measurement units. Relevant concepts from this ontology are also exposed as SKOS concepts, allowing datasets to be semantically tagged.

This dual use of OWL (for formal semantics) and SKOS (for tagging and navigation) enables powerful semantic search capabilities. For example, users can locate datasets based on vehicle subsystems, signal types, or derived relationships. The use of SKOS to map different vocabularies provides these capabilities even when different teams use inconsistent terminology.

Access Control and Provenance by Design

Unlike many data catalogs that treat governance as an external concern, this architecture *integrates governance directly into the knowledge graph*. Because datasets are explicitly modeled as both DCAT resources and PROV-O entities, provenance relationships (e.g., which activities generated a dataset, which agents were involved, etc.) are first-class citizens in the model. This is an example of what Gartner calls “active metadata” (Zaidi & Thoo, 2019).

Similarly, access control rules are expressed declaratively using ODRL and enforced consistently through a centralized data access engine. Applications interact with the data lake through stable identifiers and templates, insulating them from storage-specific details while ensuring policy compliance.

Implementation and Scale

The semantic layer is implemented using Stardog as the RDF store and reasoning engine, with SHACL used to validate data integrity constraints. At the time of reporting, the system managed nearly 2

million registered data assets, over 30 million RDF triples, and tens of petabytes of underlying data.

Why This Matters

This case study illustrates several architectural principles that generalize well beyond the automotive domain.

First, it shows that semantic technologies complement rather than replace existing big-data platforms. Storage, computation, and analytics continue to be handled by data lake technologies, while the semantic layer provides meaning, structure, and governance across heterogeneous systems. This separation of concerns allows organizations to scale operational infrastructure while maintaining uniform, automated control over semantics, provenance, governance, and access.

Second, the architecture demonstrates that event-driven systems are a natural fit for semantic metadata management. Technologies such as Apache Kafka enable ingestion and processing pipelines to emit standardized events, which in turn drive automatic population and maintenance of semantic catalogs. This pattern allows semantic metadata to remain synchronized with rapidly evolving data assets without relying on manual curation.

Third, the case shows that Semantic Knowledge Graphs can operate at true enterprise scale. Using the Stardog enterprise RDF triplestore together with standard validation mechanisms such as SHACL, the system manages millions of registered data assets, tens of millions of RDF triples, and petabytes of underlying data.

Finally, the case reinforces a central theme of this book: *governance, provenance, and access control are most effective when modeled explicitly*. By treating these concerns as first-class semantic concepts rather than as external configuration, the architecture enables transparent, queryable, and enforceable governance across a complex data ecosystem.

Taken together, these patterns demonstrate that Semantic Knowledge Graphs are foundational infrastructure for managing large-

scale, heterogeneous, and regulated data environments.

Lessons Learned

This case study highlights several lessons that are broadly applicable to enterprise-scale Semantic Knowledge Graph systems:

1. *A semantic layer should complement, not replace, existing data infrastructure*. The success of this architecture lies in treating the knowledge graph as a semantic control and discovery layer rather than as a monolithic data store. Existing big-data technologies remain responsible for storage and computation, while the semantic layer provides meaning, governance, and integration.
2. *Successful scaling requires that metadata must be populated automatically*. Manual curation does not work at enterprise scale. By integrating semantic metadata population directly into ingestion and processing pipelines, the system ensures that catalog entries, provenance, and access policies remain current as the data lake evolves.
3. *Event-driven architectures are a natural fit for semantic governance*. Publishing ingestion and processing events enables the semantic layer to react to changes in near real time. This pattern cleanly decouples operational data pipelines from semantic enrichment while maintaining strong consistency at the metadata level.
4. *Reuse of open standards pays off*. Combining DCAT, PROV-O, ODRL, and SKOS provides a rich but interoperable foundation for data cataloging, lineage, and access control. Using standardized vocabularies reduces lock-in and makes the architecture easier to evolve as requirements change. Standards provide predefined solutions to common problems that were created by the best minds in the domain and allow developers to focus on adding innovative new capabilities that give them a competitive advantage.
5. *OWL and SKOS serve complementary roles*. OWL enables expressive

conceptual domain modeling and reasoning, while SKOS supports scalable tagging, navigation, and search. Using both together avoids forcing a single modeling style to serve incompatible use cases.

6. *Governance is most effective when it is modeled explicitly*. Treating provenance and access control as first-class semantic concepts modeled using PROV-O and ODRL, respectively, makes governance transparent, queryable, and enforceable across systems.

This implementation demonstrates that semantic technologies can manage millions of assets in production environments.

Case Study 3: PatientGraph Semantic Knowledge Graphs and LLMs for Healthcare Analytics and Decision Support

The challenge: Making unstructured clinical data usable at scale

Healthcare organizations generate vast amounts of clinically relevant information. Unfortunately, most of that information is embedded in unstructured text rather than structured databases. The details that matter most for patient care and decision-making are found within clinical notes, trial descriptions, and research literature. Experience has shown that these resist conventional data integration and analytics approaches, including those based solely on large language models without explicit semantic grounding.

This case study examines how a Semantic Knowledge Graph referred to as *PatientGraph* (Bajracharya et al., 2022) was used to address this challenge by integrating structured data, unstructured text, and formal medical semantics into a single, queryable representation.

Context and Problem

Healthcare organizations face a persistent and well-known problem: most clinically relevant information is both unstructured and

distributed across clinical notes, research literature, trial descriptions, and external vocabularies. While electronic medical records (EMRs) provide structured data for billing and reporting, clinicians routinely document the most important observations in free text.

As a result, organizations struggle to answer questions that span multiple data sources, such as

- Identifying patient cohorts that meet complex eligibility criteria
- Matching patients to relevant clinical trials
- Computing clinical risk scores with traceable justification
- Combining population-level analytics with patient-level reasoning

Traditional enterprise data warehouse and analytics platforms are poorly suited to these tasks. They require rigid schemas, duplicate transformations, and extensive precomputation, making them difficult to adapt as medical knowledge and terminology evolve.

Franz Inc. addressed these challenges using AllegroGraph as the foundation for a healthcare knowledge graph architecture referred to as PatientGraph.

Why Semantic Knowledge Graphs Were Needed

Two characteristics of healthcare data made semantic technologies essential.

First, terminology heterogeneity is unavoidable. Clinical notes, trial descriptions, and research publications use different vocabularies, levels of specificity, and linguistic conventions. Even within standardized systems such as UMLS, equivalent concepts may appear under multiple labels or codes.

Second, as is true in all highly regulated environments, *reasoning and explanation capabilities are fundamental requirements*. In regulated environments, systems must be able to explain why a patient was included or excluded from a cohort, or how a particular risk score was computed.

A Semantic Knowledge Graph made it possible to

- Represent patients, events, conditions, treatments, and trials as connected entities
- Ground text-derived terms in shared medical vocabularies
- Apply logical constraints alongside statistical and language-based methods
- Preserve provenance and justification for downstream review

These requirements become even more pronounced when organizations attempt to apply large language models to healthcare data. The probabilistically generated outputs of an LLM are not acceptable—outputs must be grounded in explicit, governed domain knowledge.

Core Capabilities and Architecture

The PatientGraph approach integrates multiple types of data and inference into a single semantic layer.

At the center is a knowledge graph that connects

- Patient records and clinical events
- Standardized medical vocabularies (e.g., UMLS, SNOMED, RxNorm)
- Clinical trials and eligibility criteria
- Literature and external reporting sources

Unstructured clinical text is processed using large language models (LLMs) to extract medically relevant entities such as conditions, medications, procedures, and observations. These extracted terms are then linked to standardized concepts using a combination of

- Taxonomy-based reasoning
- Vector embedding-based matching with LLMs
- Explicit logical rules expressed in SPARQL or Prolog

This hybrid approach allows the system to handle both straightforward cases (e.g., direct matches in a vocabulary hierarchy) and more ambiguous cases where terminology differs substantially between authors or sources.

A key design decision was to combine symbolic vocabulary reasoning with vector embedding-based similarity utilized by an LLM. Taxonomies are reliable for navigating broader and narrower concepts but often fail when equivalent terms are expressed outside the hierarchy. Embeddings enable LLMs to bridge this gap by capturing contextual similarity across differently authored texts, while the knowledge graph preserves explicit semantics and provenance.

Importantly, the knowledge graph serves as a single data shape for analytics, reporting, and advanced reasoning. The same graph can support standard queries, cohort construction, rule-based inference, and integration with machine-learning models.

Example Use Case: Clinical Trial Matching

Clinical trial matching illustrates the value of this architecture particularly well.

Large healthcare providers may have tens of thousands of eligible patients and thousands of active trials at any given time. Naively matching every patient to every trial using LLM-based comparison is computationally expensive and impractical at scale.

Rather than treating clinical trial matching as a single monolithic problem, the PatientGraph architecture decomposes it into a small number of semantically distinct steps. Each step is designed to aggressively reduce the search space while preserving recall, ensuring that clinically relevant trials are not excluded prematurely.

This decomposition reflects four key design principles:

1. Use clinical notes as the primary source of truth since they contain the majority of clinically relevant detail.
2. Use structured EMR data selectively, primarily for coarse filtering (e.g., age, gender, high-level conditions).
3. Generate richer semantic annotations using LLMs, because simple entity extraction is insufficient for complex eligibility criteria.
4. Combine taxonomy-based reasoning with embedding-based

matching and explicit rules, allowing the system to handle both standardized and highly variable medical language.

These principles are implemented operationally as follows:

- Structured EMR data is used for coarse pre-filtering (e.g., age, gender, primary conditions).
- Clinical notes are treated as the primary source of truth for detailed eligibility criteria.
- LLMs generate rich annotations from both clinical notes and trial descriptions.
- Semantic reasoning and rule-based constraints dramatically reduce the candidate space before invoking more expensive language-model comparisons.

The result is a system that preserves recall for critical trials while reducing cost, latency, and cognitive burden on clinicians.

Outcomes and Lessons Learned

Several general lessons emerge from this case study.

- Semantic grounding is essential when working with heterogeneous medical language. Embeddings alone are insufficient without explicit conceptual structure because of hallucinations and black-box reasoning.
- Explanation matters as much as prediction in healthcare. Logical constraints and graph structure provide transparency that purely statistical systems lack.
- One semantic layer can support many use cases, from reporting and cohort analysis to decision support and AI-assisted workflows.
- Future-proofing is practical, not theoretical. As vocabularies, regulations, and models evolve, the knowledge graph can adapt without wholesale redesign.

Why This Matters

This case study demonstrates how Semantic Knowledge Graphs enable healthcare organizations to move beyond siloed analytics and brittle

pipelines. By unifying structured data, unstructured text, and formal semantics in a single graph, systems become easier to reason over, easier to explain, and easier to evolve.

More broadly, it illustrates a recurring theme throughout this book: Semantic Knowledge Graphs are not a replacement for machine learning or language models, but a complement to them. This case illustrates the broader theme of this book: semantic reasoning and language models are most effective when combined.

Case Study 4: Avalara—Using Graphwise Neurosymbolic Architecture to Implement GraphRAG

The challenge: Utilizing LLM customer support in a regulated domain

Avalara provides tax compliance and automation services. Their global business provides features such as automated tax compliance, calculation, and reporting services in a domain where correctness is critical. Errors are not merely inconvenient; they can result in regulatory violations, financial penalties, and loss of customer trust.

Like many organizations, Avalara explored large language models to improve customer support and knowledge access. However, early experiments revealed a fundamental limitation: LLMs are probabilistic systems that can generate plausible but incorrect responses known as hallucinations. Hallucinations are unacceptable in just about any business system but even more so in a regulated environment.

The Graphwise GraphRAG Solution: Separating Knowledge and Communication

Avalara first attempted to address the problem using a conventional RAG architecture with a corpus stored in a traditional data store. However, this first RAG implementation lacked the accuracy and

explicit knowledge representation of a Semantic Knowledge Graph. This resulted in what Avalara described as a “Precision Paradox,” where even small errors (despite high overall accuracy) led to user dissatisfaction.

Avalara addressed this problem by adopting a graph retrieval-augmented generation (GraphRAG) system, built on a neurosymbolic foundation combining knowledge graphs and vector-based language models (Graphwise, 2025b).

In this architecture, the knowledge graph serves as the authoritative semantic layer, while the language model serves as a linguistic interface rather than a knowledge base. This separation allows the system to benefit from the expressive power of large language models to communicate via natural language, while providing the explicit knowledge representation of a Semantic Knowledge Graph. This explicit knowledge representation provided correctness, traceability, and governance in a way that the black-box architecture of a standalone LLM and the shallow knowledge representation of a traditional RAG are incapable of.

Key architectural elements included

- A knowledge graph derived from structured, curated documentation, defined using the Darwin Information Typing Architecture (DITA), an XML-based standard for topic-oriented technical documentation. The use of DITA was significant because it provided a pre-existing, governed, and semantically structured knowledge source, reducing the risk of introducing unvetted or inconsistent information into the AI system.
- Automated conversion from DITA to RDF, preserving content relationships and hierarchical structure
- Explicit modeling of domain concepts and their relationships
- Retrieval based upon, and justified by, semantic structure rather than mere raw similarity
- LLMs used to interpret questions and generate responses that are grounded in verified knowledge from the corpus and Semantic Knowledge Graph

The transition to a Semantic Knowledge Graph was not merely an implementation change, but a shift from shallow, implicit semantics to explicit, governed domain knowledge. This architecture deliberately separates *what the system knows* from *how it communicates*. The knowledge graph provides determinism, traceability, and governance. The LLM provides flexibility and linguistic fluency.

Outcomes and Impact

The GraphRAG approach produced several critical benefits:

- Substantial reduction in hallucinated responses
- Improved explainability and traceability of answers
- Greater confidence deploying AI in customer-facing systems
- A reusable semantic foundation for future AI applications

Rather than replacing knowledge engineering, the use of LLMs has increased its importance. The more powerful the language model, the more essential it becomes to constrain it with explicit domain knowledge.

Why This Matters

This case illustrates a broader shift in enterprise AI architectures. LLMs are most effective when paired with structured, governed knowledge sources. Semantic Knowledge Graphs provide this capability in a form that is both human-understandable and machine-readable. Knowledge graphs are enabling infrastructure for successful deployment of LLMs.

Case Study 5: Semantic Knowledge Graphs for Essilor Global Supply Chain Risk Management

The challenge: Managing risk in highly interconnected global supply chains

Modern supply chains are deeply interconnected systems that span thousands of suppliers, multiple tiers of dependencies, and geopolitical boundaries. A disruption in one region can rapidly cascade across the

entire network, causing manufacturing delays, cost overruns, and operational failures. Disruptions can be caused by many diverse events such as natural disasters, political instability, sanctions, or supplier failure.

Traditional supply chain risk management systems struggle to address this complexity. They typically provide only partial visibility into supplier relationships, model dependencies at limited depth, and rely on rigid relational schemas that are poorly suited to represent evolving, many-to-many relationships. As a result, organizations often discover critical vulnerabilities only after disruptions occur.

This case study examines how a Semantic Knowledge Graph architecture built on the AllegroGraph platform was used by Essilor to model, analyze, and predict supply chain risk in large, mission-critical supply networks (Franz Inc., 2021).

Context and Problem

Large manufacturers and defense contractors depend on supply chains that may involve tens of thousands of components sourced from suppliers distributed across the globe. These components are often shared across multiple products and programs, and suppliers themselves may depend on other suppliers several tiers removed from the original manufacturer.

Several long-term industry trends have amplified supply chain risk:

- Increased outsourcing and offshoring
- Reliance on single-source suppliers for specialized components
- Just-in-time delivery models
- Lean inventory strategies

While these practices improve efficiency, they also increase fragility (sometimes drastically). Events such as semiconductor shortages, geopolitical sanctions, natural disasters, or regional conflicts can disrupt production for months or years.

Existing supply chain risk management tools are typically based on relational databases and business-process workflows. These systems

lack the expressive power needed to represent deep dependency chains, rapidly evolving relationships, and the dynamic impact of external events.

Why Semantic Knowledge Graphs Were Needed

Supply chain risk is ***fundamentally a graph problem***.

A single component may be used in many products. A single supplier may serve many manufacturers. An external event such as a natural disaster or political action may affect regions, suppliers, logistics, and regulatory constraints simultaneously.

Semantic Knowledge Graphs are well suited to this domain because they can

- Model many-to-many relationships naturally
- Represent entities, events, and dependencies explicitly
- Evolve schemas without disrupting existing systems
- Integrate heterogeneous data sources incrementally

In this architecture, the knowledge graph does not replace existing operational systems. Instead, it acts as a semantic integration and reasoning layer that aggregates "master data" from multiple supply-chain, inventory, and logistics systems into a unified representation.

Core Capabilities and Architecture

The supply chain risk management architecture is built around a Semantic Knowledge Graph that represents

- Products and components
- Suppliers and sub-suppliers
- Geographic regions and facilities
- Logistics and transportation relationships
- External events such as natural disasters, sanctions, or geopolitical incidents

Data is continuously ingested from multiple operational systems using change-data-capture and streaming techniques. Only the data relevant to risk analysis is extracted, transformed into a unified

semantic model, and loaded into the knowledge graph. This approach avoids disruptive “big bang” migrations and allows the system to operate continuously alongside existing infrastructure.

To assess risk, the supply chain knowledge graph is federated with external event databases that capture real-world incidents. This enables queries and analytics that reason over both structural dependencies and unfolding events.

Traditional machine-learning techniques can then be applied by generating tabular features from SPARQL queries over the graph. In addition, graph-native machine-learning approaches such as Graph Neural Networks (GNNs) can be used to analyze how events may propagate through the supply chain network and to estimate the likelihood and impact of disruptions.

Example Use Case: Supply Chain Risk Assessment

Risk assessment in this system focuses on identifying how likely events may affect critical suppliers, components, or regions.

For example, the knowledge graph can be queried to determine

- Which products depend on a supplier located in a high-risk region
- Which alternative suppliers exist several tiers removed
- How disruptions might cascade across programs or product lines

By combining explicit dependency modeling with event data and machine-learning analysis, the system supports proactive risk identification rather than reactive response. Decision-makers can explore “what-if” scenarios, assess potential impacts, and prioritize mitigation strategies before disruptions materialize.

Outcomes and Lessons Learned

This case study highlights several lessons relevant to large-scale enterprise systems:

- *Visibility is a prerequisite for risk management*. Without an explicit model of supply chain dependencies, risk cannot be analyzed effectively.

- *Relational models are often poorly suited to representing deep, evolving dependency graphs*. Many-to-many relationships and evolving structures are more naturally represented in graphs.
- *Semantic integration enables incremental modernization*. Existing systems can remain in place while semantic models unify their data.
- *Events matter as first-class concepts*. Incorporating real-world events directly into the knowledge graph enables more realistic risk assessment.

Supply chain risk management illustrates a broader point about Semantic Knowledge Graphs: *they excel in domains where structure, change, and uncertainty intersect*. By making dependencies explicit and queryable, knowledge graphs enable organizations to reason about complex systems that would otherwise remain opaque.

In this case, Semantic Knowledge Graphs provide the foundation for integrating operational data, external events, and machine-learning analysis into a single, coherent framework. The result is not merely better analytics, but a more resilient and adaptable enterprise architecture.

Summary

These case studies illustrate some of the most important points about Semantic Knowledge Graphs described in this book:

- *Semantic Knowledge Graphs provide a foundation for enterprise management of big data*. They provide real value not as standalone artifacts but when integrated with enterprise data and systems.
- *Semantic Knowledge Graphs don't compete with large language models and other machine learning models*. On the contrary, they complement such models. The strengths of semantic technology complement the weaknesses of LLMs and vice versa.
- *Semantic Knowledge Graphs are a natural fit for the event-driven architectures used by the most forward-thinking organizations for*

microservices and data products. They integrate very well and bring added value to technologies such as Kafka, Docker, and Kubernetes.

- *Semantic Knowledge Graphs are an essential foundation for developing analytic architectures which satisfy the needs of Data Mesh and Data Fabrics*, e.g., by enabling active metadata, graph modeling, and explicit knowledge representation of semantic metadata.
- *Further Reading*: Although it is a bit dated (written before the maturity of LLMs), the *Knowledge Graph Cookbook* (Blumauer & Nagy, 2020) is an excellent resource for examples of applications developed using Semantic Knowledge Graphs that have been deployed and shown significant value.
- *Further Reading*: The GitHub site for this book has additional examples, tutorials, code, etc. It will continue to be updated after the book has been published. Best places to start are the ReadMe page https://github.com/mdebellis/SemanticKG-Design and the Wiki home page https://github.com/mdebellis/SemanticKG-Design/wiki.
- *Further Reading*: My blog focuses primarily on papers, opinions, best practices, etc., mostly related to the Semantic Web and LLMs: https://www.michaeldebellis.com/blog.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
M. DeBellis, *Designing Semantic Knowledge Graphs*
https://doi.org/10.1007/979-8-8688-1823-3_15

15. Conclusion from Models to Systems

Michael DeBellis[1] ✉
(1) San Francisco, CA, USA

This book began with a simple premise: *Semantic technologies are not a niche research area. They are the most mature and technically grounded foundation available for modern enterprise data architectures*.

Over the past decade, industry leaders have introduced powerful architectural movements: Data Mesh, Data Fabric, Data Lakehouse, Data Catalogs, knowledge management platforms, and event-driven microservices. Across these approaches, certain themes repeatedly emerge: active metadata, governance as code, graph modeling, globally unique identifiers, machine-readable contracts, and explicit semantics.

Yet in most of these discussions, the technologies designed specifically to address these challenges—RDF, OWL, SHACL, SPARQL, IRIs—are rarely mentioned. This book argues that this coverage gap is not technical. It is translational. The communities which have developed these technologies and the software engineering communities which could benefit from them have limited overlap. This book seeks to bridge that gap.

Semantic Technologies As Enterprise Infrastructure

Semantic technologies provide

- Active metadata that is executable and queryable

- Semantic models that separate conceptual design from physical storage
- Governance as code, enforced through constraints and validation
- Graph-native modeling and storage, aligned with real-world relationships
- IRIs as persistent identifiers, enabling global integration

These are the capabilities enterprise architects describe when they speak about composable data platforms. Many of these challenges have long-standing solutions within the Semantic Web standards. What has been missing is a comprehensive effort to convey the technical maturity of semantic modeling in the language of enterprise software engineering. That is the primary goal of this book.

Ontologies and Large Language Models Are Not Competitors

Another central theme of this book has been the relationship between ontologies and large language models (LLMs). Some view LLMs as a threat to symbolic approaches. Others view them as eliminating the need for symbolic approaches. Both perspectives misunderstand their complementary nature, strengths, and weaknesses.

LLMs excel at pattern recognition, language generation, and probabilistic inference. They do not model or store data. They do not persist data or metadata. They do not maintain identifiers, enforce constraints, guarantee consistency, or track provenance and ownership.

As LLMs become more central to enterprise systems, the need for explicit semantics becomes greater, not smaller. GraphRAG, active metadata, semantic data catalogs, and structured prompts all demonstrate that ontologies and LLMs are complementary. The most robust systems combine statistical intelligence with explicit knowledge representation.

The future is hybrid.

From Academic Exercise to Working System

An ontology without data, or without supporting code, is an academic exercise. Enterprise systems require more.

Throughout this book, we have emphasized ingestion pipelines, validation layers, SPARQL transformations, SHACL constraints, integration with Python, and interaction with production architectures and tools such as Kafka and BigQuery. We have tried to show how to work with flawed legacy data, evolving models, and real governance constraints.

Modeling is the easy part. Integration is the hard part. This is one of the first things I learned when I began my career as a professional developer developing rule- and LISP-based systems for clients of Accenture (then Andersen Consulting) as a member of the Andersen AI group in the 1980s. Defining the rules was the easy part. However, if all we did was create a rule-based system or an ontology running on a PC or LISP machine, it was never more than a prototype. The real systems had to integrate with existing databases, mainframe systems in COBOL, and enterprise applications such as SAP. That was the hard part.

The languages and models have changed, but the basic facts of software engineering haven't. *Successful enterprise systems depend not only on conceptual clarity, but on usability, interoperability, performance, and incremental delivery.*

Rejecting False Dichotomies

This book takes positions that differ from some traditional ontology approaches.

It does not treat ontology design as a philosophical exercise. Upper ontologies and foundational theories have value in domains like biomedical research where understanding is a goal in itself. However, enterprise software development operates under different constraints: evolving requirements, distributed teams, performance limitations, iterative development, and user-centered design. It does not assume models must be perfect before deployment.

The ontology design mindset of “get it right the first time” mirrors the waterfall model in software engineering—a model that industry largely abandoned decades ago. The impression that this mindset is all that there is to ontologies has become far and away the biggest obstacle to recognition of the value ontologies offer when used with an engineering mindset.

Semantic systems benefit from iterative refinement, prototyping, and continuous feedback from users.

They do not dismiss semiformal structures, such as taxonomies and glossaries, that capture how users actually think. When developed with SKOS, and integrated thoughtfully, semiformal techniques provide intuitive entry points and the foundation for formal ontologies. They also support UI design, navigation, and discovery.

Again, the future is hybrid. *Enterprise semantics requires an engineering mindset that is eclectic and that combines rigor with pragmatism.*

What This Book Does Differently

Many ontology books focus on modeling patterns in abstraction. This book goes further.

It shows how to

- Design usable OWL models
- Validate them with SHACL
- Query and transform them with SPARQL
- Integrate them with programming languages and distributed computing APIs
- Load and evolve real data
- Connect them to Data Mesh concepts, data products, and Domain-Driven Design
- Integrate with large language models

It grounds theory in implementation. The goal is not merely to understand semantic technologies, but to deploy them.

Many software systems engineering methodologies focus on practices and do not push the boundaries on supporting technology. This book goes further.

It shows how semantic techniques can effectively implement and execute engineering principles of active metadata, governance as code, graph modeling, globally unique identifiers, machine-readable contracts, and explicit semantics. Further, it provides illustrations of the potential for time and cost savings in the creation and maintenance of large, data-intensive enterprise information systems.

The Path Forward

We are at an inflection point. Enterprises are overwhelmed with data. They are investing heavily in AI, but many feel that these investments have failed to meet expectations. They are redesigning their architectures to be domain-oriented, federated, and metadata-driven.

Semantic technologies are uniquely positioned to provide the connective tissue that binds these systems together. For that to happen, we must speak the language of software engineers. We must demonstrate integration, not only modeling. We must show value in production systems.

The next generation of enterprise platforms that combine explicit semantics, active metadata, Domain-Driven Design, and LLM-assisted reasoning will not be built by choosing between symbolic and statistical approaches. They will be built by integrating them.

This next generation will reposition semantic technologies as what they have always had the potential to be: *foundational infrastructure for intelligent, governed, and evolvable enterprise systems and practical tools for the engineers who build them.*

Bibliography

Aasman, J. & Cheatham, K., 2011. *RDF browser for data discovery and visual query building.* Palo Alto, CA, USA, s.n.

Albertoni, R. et al., 2024. *Data Catalog Vocabulary (DCAT) - Version 3.* [Online] Available at: https://www.w3.org/TR/vocab-dcat-3/ [Accessed 13.1.2026].

Allemang, D. & Hendler, J., 2011. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL.* Waltham, MA, USA: Morgan Kaufmann.

Anastasiu, D. C. & Karypis, G., 2016. *Fast Parallel Cosine K-Nearest Neighbor Graph Construction.* Salt Lake City, UT, USA, s.n., pp. 50-53.

Andrews, H. et al., 2025. *OpenAPI Specification v3.2.0.* [Online] Available at: https://spec.openapis.org/oas/v3.2.0.html [Accessed 14.5.2026].

Apache Software Foundation, 2025. *Jena Ontology API.* [Online] Available at: https://jena.apache.org/documentation/ontology/ [Accessed 17.11.2025].

Arp, R., Smith, B. & Spear, A. D., 2015. *Building Ontologies with Basic Formal Ontology.* Cambridge, MA, USA: MIT Press.

ART Research Group, 2025. *VocBench.* [Online] Available at: https://vocbench.uniroma2.it/ [Accessed 10.12.2026].

Ashburner, M. et al., 2000. Gene Ontology: tool for the unification of biology. *Nature Genetics,* May, Volume 25, p. 25–29.

Baader, F. et al., 2003. *The Description Logic Handbook.* Cambridge: Cambridge University Press.

Bajracharya, R., Wallace, R., Aasman, J. & Mirhaji, P., 2022. *Entity Event Knowledge Graph for Powerful Health Informatics.* Rochester, MN, USA., IEEE.

Beck, K., 2004. *Extreme Programming Explained.* Boston, MA, USA: Addison-Wesley.

Berners-Lee, T., Hendler, J. & Lassila, W. O., 2001. The Semantic Web: A new form of Web content that is meaningful to computers will

unleash a revolution of new possibilities. *Scientific American*, 17 May.

Bird, S., Klein, E. & Loper, E., 2009. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit.* Sebastapol, CA, USA: O'Reilly.

Blackwood, P., 2020. *A Brief Introduction to the Gist Semantic Model.* [Online] Available at: https://www.semanticarts.com/a-brief-introduction-to-the-gist-semantic-model/ [Accessed 19.2.2025].

Blumauer, A. & Nagy, H., 2020. *The Knowledge Graph Cookbook: Recipes That Work.* [Online] Available at: https://www.poolparty.biz/the-knowledge-graph-cookbook [Accessed 13.2.2026].

Boehm, B., 1986. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes,* August, 11(4), pp. 14-24.

Brooks, F., 1987. No Silver Bullet Essence and Accidents of Software Engineering. *IEEE Computer,* 20(4), pp. 10-19.

Chomsky, N., 2016. *What Kind of Creatures Are We?.* Chichester, New York: Columbia University Press.

Clark, K., 2024. *Enterprise AI Requires the Fusion of LLM and Knowledge Graph.* [Online] Available at: https://www.stardog.com/blog/enterprise-ai-requires-the-fusion-of-llm-and-knowledge-graph/ [Accessed 5.1.2026].

Conway, M., 1999. *Conways Law.* [Online] Available at: https://www.melconway.com/Home/Conways_Law.html [Accessed 7.4.2026].

DeBellis, M., 2020. *What's in a Name? IRIs, Labels, and UUIDs.* [Online] Available at: https://www.michaeldebellis.com/post/names-iris-labels-and-uuids [Accessed 25.3.2025].

DeBellis, M., 2021. *New Protégé Pizza Tutorial.* [Online] Available at: https://www.michaeldebellis.com/post/new-protege-

pizza-tutorial [Accessed 11.3.2024].

DeBellis, M., 2023a. *Puns in the Web Ontology Language (OWL): Why We Need Them and How to Use Them.* [Online] Available at: https://www.michaeldebellis.com/post/puns_in_owl_the_whys_and_hows [Accessed 30.1.2026].

DeBellis, M., 2023. *Implementing the Allen Model with the W3C Time Ontology and SWRL Rules.* [Online] Available at: https://github.com/mdebellis/Time-Ontology-With-SWRL-Rules [Accessed 16.1.2025].

DeBellis, M., 2024a. *Using ChatGPT to Generate English Labels for a Spanish Ontology.* [Online] Available at: https://www.michaeldebellis.com/post/generating_english_ontology_labels [Accessed 9.1.2026].

DeBellis, M., 2024. *The Turing/Church Proofs: How Logicians Created Computer Science.* [Online] Available at: https://www.michaeldebellis.com/post/the-turing-church-proof [Accessed 19.3.2025].

DeBellis, M., 2025. *Basic Reusable Ontology (BRO).* [Online] Available at: https://github.com/mdebellis/Basic_Reusable_Ontology [Accessed 8.12.2025].

DeBellis, M., 2026. *Communicating Ontology Knowledge with an LLM in Graph RAG: The power of simplicity and standards.* [Online] Available at: https://www.michaeldebellis.com/post/communicating-ontology-knowledge-with-an-llm-in-graph-rag-the-power-of-simplicity-and-standards [Accessed 13.5.2026].

DeBellis, M. & Dutta, B., 2022. From ontology to knowledge graph with agile methods: the case of COVID-19 CODO knowledge graph. *International Journal of Web Information Systems,* 5 October.18(5/6).

DeBellis, M., Dutta, N., Gino, G. & Balaji, A., 2024. Integrating Ontologies and LLMs to Implement Retrieval Augmented Generation (RAG). *Applied Ontology.*

DeBellis, M. et al., 2023. *DaanKG: An Ontology Model of the UN Sustainable Development Goals to Facilitate and Improve Corporate Social Responsibility.* Sherbrook, Quebec Ontario, s.n.

DeBellis, M., Gino, G., Balajic, A. & Gino, J., 2025. *Modeling Climate Obstruction with a Neurosymbolic Architecture.* Catania, Italy, IOS Press.

DeBellis, M., Gino, G., Balajic, A. & Gino, J., 2025. *Using Retrieval Augmented Generation (RAG) and Knowledge Graphs to Understand Climate Obstruction.* Boulder, s.n.

DeBellis, M. & Neches, R., 2023. Knowledge Representation and The Semantic Web: An HistoricalOverview of Influences on Emerging Tools. *Recent Advances in Computer Science and Communications,* 16(6).

DeBellis, M., Pinera, L. & Connor, C., 2023. Interoperability Frameworks: Data Fabric and Data Mesh Architectures. In: D. N. C. Debnath & D. A. Patel, eds. *Data Science with Semantic Technologies.* Boca Raton(Florida): CRC Press.

Dehghani, Z., 2020. *Data Mesh Principles and Logical Architecture.* [Online] Available at: https://martinfowler.com/articles/data-mesh-principles.html [Accessed 16.2.2026].

Dehghani, Z., 2022. *Data Mesh.* Sebastopol, California, USA: O'Reilly.

Dibowski, H. & Schmid, S., 2021. Using knowledge graphs to manage a data lake. *INFORMATIK 2020,* pp. 41-50.

Dibowski, H. et al., 2020. *Using Semantic Technologies to Manage a Data Lake: Data Catalog, Provenance and Access Control.* Athens, Greece, s.n.

Dublin Core Metadata Initiative (DCMI), 2024. *The Bibliographic Ontology (BIBO).* [Online] Available at: https://www.dublincore.org/specifications/bibo/ [Accessed 16.4.2024].

DuCharme, B., 2013. *Learning SPARQL.* Sebastopol, CA, USA: O'Reilly.

Eclipse Foundation, Inc., 5. *Eclipse RDF4J.* [Online] Available at: https://rdf4j.org/ [Accessed 17.11.2025].

EDM Council, 2024. *The Financial Industry Business Ontology.* [Online] Available at: https://spec.edmcouncil.org/fibo/

[Accessed 30.4.2024].

Eryurek, E. et al., 2021. *Data Governance: The Definitive Guide: People, Processes, and Tools to Operationalize Data Trustworthiness.* Sevastapol, CA, USA: O'Reilly Media.

Evans, E., 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* San Francisco, CA, USA: Addison-Wesley.

explosion.ai, 2025. *spaCy EntityRecognizer Class.* [Online] Available at: https://spacy.io/api/entityrecognizer [Accessed 15.7.2025].

Fowler, M., 1997. *Analysis Patterns: Reusable Object Models.* New York: Addison-Wesley.

Fowler, M., 2002. *Patterns of Enterprise Application Architecture.* Reading(MA): Addison-Wesley Professional.

Fowler, M. & Lewis, J., 2014. *Microservices: A Definition of This New Architectureal Term.* [Online] Available at: https://martinfowler.com/articles/microservices.html [Accessed 16.2.2026].

Franz Inc., 2021. *Essilor Uses AllegroGraph to Build a Knowledge Graph Solution for Global Supply Chain Risk Management.* [Online] Available at: https://allegrograph.com/press_room/essilor-uses-allegrograph-to-build-a-knowledge-graph-solution-for-global-supply-chain-risk-management/ [Accessed 2.2.2026].

Franz Inc., 2024. *AllegroGraph 8.1.1 LLM Examples.* [Online] Available at: https://franz.com/agraph/support/documentation/current/llm-examples.html [Accessed 24.5.2024].

Franz Inc., 2026a. *GraphTalker: The AI Agent for your Knowledge Graph.* [Online] Available at: https://allegrograph.com/products/graphtalker-ag/ [Accessed 20.5.2026].

Franz Inc., 2026. *GraphTalker: AI Interface for Your Graph.* [Online] Available at: https://franz.com/agraph/support/documentation/graphtalker.html [Accessed 12.5.2026].

Franz, Inc., 2025. *Distributed and Federated Repositories.* [Online] Available at: https://franz.com/agraph/support/documentation/agraph-introduction.html#Distributed-and-Federated-Repositories [Accessed 26.1.2026].

Gilbert, S. & Lynch, N., 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News,* 33(2), p. 51–59.

graphql.org, 2026. *Introduction to GraphQL.* [Online] Available at: https://graphql.org/learn/introduction/ [Accessed 6.5.2026].

Graphwise, 2025a. *Scaling Search and Content Governance with Semantic Technology.* [Online] Available at: https://graphwise.ai/success-story/microsoft-docs-scaling-search-and-content-governance-with-semantic-technology/ [Accessed 2.2.2026].

Graphwise, 2025b. *Transforming AI Reliability by Building Knowledge Graph-Powered Customer Support.* [Online] Available at: https://graphwise.ai/success-story/avalara-transforming-ai-reliability-by-building-knowledge-graph-powered-customer-support/ [Accessed 2.2.2026].

Graphwise, 2025. *Graphwise.* [Online] Available at: https://graphwise.ai/ [Accessed 10.12.2025].

Graphwise, 2026. *Semantic Analytics: Graph-based text mining at the highest level.* [Online] Available at: https://graphwise.ai/components/graph-and-text-analytics/ [Accessed 12.2.2026].

Green, C. et al., 1983. *Report on a Knowledge Based Software Assistant,* Palo Alto, CA, USA: Kestrel Institute.

Gruber, T., 1993. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal Human-Computer Studies,* 23 August , Volume 43, pp. 907-928.

Gupta, L., 2025. *REST API Tutorial.* [Online] Available at: https://restfulapi.net/ [Accessed 6.5.2025].

Harris, S. & Seaborne, A., 2013. *SPARQL 1.1 Query Language W3C Recommendation,* Cambridge, MA USA: W3C.

Highsmith, J. A., 1999. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* New York, NY: Dorset House.

Horrocks, I. et al., 2004. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* [Online] Available at: https://www.w3.org/submissions/SWRL/#8.2 [Accessed 13.10.2025].

Hunger, M., Boyd, R. & Lyon, W., 2021. *The Definitive Guide to Graph Databases for the RDBMS Developer.* San Mateo, CA, USA: Neo Technology.

International Organization for Standardization, 2026. *ISO: Global standards for trusted goods and services.* [Online] Available at: https://www.iso.org/home.html [Accessed 13.1.2026].

Jackson, R. C. et al., 2019. ROBOT: A Tool for Automating Ontology Workflows. *BMC Bioinformatics,* 20(407).

Jackson, R. et al., 2021. OBO Foundry in 2021: operationalizing open data principles to evaluate ontologies. *Database,* Volume 2021.

Knublauch, H., 2026. *SHACL PlaygroundA constraint validator for the Shapes Constraint Language.* [Online] Available at: https://shacl.org/playground/ [Accessed 13.1.2026].

Knublauch, H. & Kontokostas, D., 2017. *Shapes Constraint Language (SHACL): W3C Recommendation.* [Online] Available at: https://www.w3.org/TR/shacl/ [Accessed 5.3.2025].

Lakoff, G., 1987. *Women, Fire, and Dangerous Things.* Chicago, IL: The University of Chicago Press.

LAMY, J.-B., 2023. *Welcome to Owlready2's Documentation!.* [Online] Available at: https://owlready2.readthedocs.io/ [Accessed

17.11.2025].

Lewis, H. R. & Papadimitriou, H. C., 1981. *Elements of the Theory of Computation.* Englewood Cliffs, New Jersey, USA: Prentice Hall.

Lewis, P. et al., 2020. *Retrieval-augmented generation for knowledge-intensive NLP tasks.* Vancouver, Ontaria, Canada, Curran Associates Inc., p. 9459–9474.

McComb, D., 2019. *The Data-Centric Revolution: Restoring Sanity to Enterprise Information Systems.* Sedona, AZ, USA: Technics Publications.

McRoberts, M. & Doncel, V. R., 2014. *Open Digital Rights Language (ODRL) Ontology.* [Online] Available at: https://www.w3.org/ns/odrl/2/ODRL20.html [Accessed 17.11.2025].

Menon, P., 2022. *Data Lakehouse in Action: architecting a modern and scalable data analytics platform.* Birmingham, UK: 2022 Packt Publishing.

Meyer, B., 1988. *Object-oriented Software Construction.* New York, NY, USA: Prentice Hall.

Michel Dumontier, R. Ç. K. G. I. d. Z., Serafimova, K., Costa, C. M. & Schulz, S., 2025. *SULO - a simplified upper-level ontology.* Catania, Italy, International Conference on Formal Ontology in Information Systems (FOIS 2025).

Mikolov, T., Chen, K., Corrado, G. & Dean, J., 2013. *Efficient Estimation of Word Representations in Vector Space.* [Online] Available at: https://arxiv.org/abs/1301.3781 [Accessed 8.12.2025].

München, E.-B. a. W. S. R. G. U. d. B., 2025. *Dr. Martin Hepp GoodRelations: The Web Vocabulary for E-Commerce.* [Online] Available at: https://www.heppnetz.de/projects/goodrelations/ [Accessed 31.3.2026].

Neches, R. et al., FALL 1991. Enabling Technology for Knowledge Sharing. *AI Magazine,* 12(3).

Neel Nanda et. al., 2023. *Fact Finding: Attempting to Reverse-Engineer Factual Recall on the Neuron Level.* [Online] Available at: https://www.alignmentforum.org/posts/iGuwZTHWb6DFY

3sKB/fact-finding-attempting-to-reverse-engineer-factual-recall [Accessed 26.12.2024].

Nielsen, J., 2024. *10 Usability Heuristics for User Interface Design.* [Online] Available at: https://www.nngroup.com/articles/ten-usability-heuristics/ [Accessed 28.4.2026].

OBO Foundry, 2026. *ROBOT is an OBO Tool.* [Online] Available at: https://robot.obolibrary.org/ [Accessed 14.1.2026].

OBO Technical WG, 2024. *OBO Principles: Overview.* [Online] Available at: https://obofoundry.org/principles/fp-000-summary.html [Accessed 9.1.2026].

ODRL Community Group, 2018. *ODRL Information Model 2.2.* [Online] Available at: https://www.w3.org/TR/odrl-model/ [Accessed 10.2.2026].

Olesen-Bagneux, O., 2022. *The most important data catalog capability?.* [Online] Available at: https://www.linkedin.com/pulse/most-important-data-catalog-capability-ole-olesen-bagneux/ [Accessed 8.2.2026].

Olesen-Bagneux, O., 2023. *The Enterprise Data Catalog.* Sebastopol, CA, USA: O'Reilly.

OpenAPI Initiative. (2026). *OpenAPI Specification v3.2.1* (10 September 2026). https://spec.openapis.org/oas/v3.2.1.html.

Open University, 2025. *NeOn Toolkit (Archived Research).* [Online] Available at: https://research-archive.stem.open.ac.uk/neon-toolkit/ [Accessed 10.12.2025].

Overton, J. et al., 2021. *OBO Semantic Engineering Training.* [Online] Available at: https://oboacademy.github.io/obook/ [Accessed 9.1.2026].

Pan, J. Z. & Uschold, M., 2006. *A Semantic Web Primer for Object-Oriented Software Developers.* [Online] Available at:

https://www.w3.org/TR/sw-oosd-primer/ [Accessed 13.2.2026].

Popov, P., 2025. *The Advantages of GraphRAG for Enhanced Regulatory Compliance and Understanding.* [Online] Available at: https://graphwise.ai/blog/the-advantages-of-graphrag-for-enhanced-regulatory-compliance-and-understanding/ [Accessed 5.1.2026].

Poveda-Villalón, M., Gómez-Pérez, A., Suárez-Figueroa & Carmen, M., 2014. OOPS! (Ontology Pitfall Scanner!): An on-line tool for ontology evaluation.. *International Journal on Semantic Web and Information Systems (IJSWIS),* 10(2), pp. 7-34.

Prov W3C Working Group, 2013. *PROV-Overview.* [Online] Available at: https://www.w3.org/TR/2013/NOTE-prov-overview-20130430/ [Accessed 8.5.2023].

Ray, S., 2025. *QUDT - Quantities, Units, Dimensions and DataTypes.* [Online] Available at: https://github.com/qudt/qudt-public-repo [Accessed 9.1.2026].

RDFLib Open Source Community, 2025. *RDFLib Home.* [Online] Available at: https://rdflib.dev/ [Accessed 17.11.2025].

Richardson, L., 2007. *Beautiful Soup Documentation.* [Online] Available at: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ [Accessed 17.4.2024].

Sheth, A., Roy, K. & Gaur, M., 2023. Neurosymbolic AI -- Why, What, and How. *IEEE Intelligent Systems.*

Singhal, A., 2012. *Introducing the Knowledge Graph: things, not strings.* [Online] Available at: https://www.blog.google/products/search/introducing-knowledge-graph-things-not/ [Accessed 8.5.2023].

SNOMED International , 2024. *SNOMED Governance and Advisory.* [Online] Available at: https://www.snomed.org/governance-and-advisory [Accessed 29.4.2024].

Snowflake Inc., 2024. *Streamlit IO Home Page.* [Online] Available at: https://streamlit.io/ [Accessed 23.2.2024].

Strengholt, P., 2023. *Data Management at Scale: Modern Data Architecture with Data Mesh and Data Fabric.* Sevastapol(CA): O'Reilly Media.

Talisman, J., 2025. *The Ontology Pipeline.* [Online] Available at: https://jessicatalisman.substack.com/p/the-ontology-pipeline [Accessed 21.11.2026].

The Dublin Core™ Metadata Initiative (DCMI), 2024. *DCMI Metadata Terms.* [Online] Available at: https://www.dublincore.org/specifications/dublin-core/dcmi-terms/ [Accessed 16.4.2024].

Top Quadrant Inc., 2025. *Top Quadrant.* [Online] Available at: https://www.topquadrant.com/ [Accessed 10.12.2025].

Tudorache, Tania, et al., 2020. WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the Web. *Semantic Web,* 11(4), p. 485–504.

Uschold, M., 2018. *Demystifying Owl for the Enterprise (Synthesis Lectures on Semantic Web: Theory and Technology).* San Rafael, California (USA): Morgan & Claypool.

W3C OWL Working Group, 2012. *OWL 2 Web Ontology Language Document Overview (Second Edition),* Sophia-Antipolis, France: W3C.

W3C Schema.org Community Group, 2026. *Schema.org.* [Online] Available at: https://schema.org/ [Accessed 31.3.2026].

W3C SKOS Working Group, 2005. *SKOS Core Vocabulary Specification.* [Online] Available at: https://www.w3.org/TR/swbp-skos-core-spec/ [Accessed 5.12.2024].

W3C, 2012. *OWL 2 Web Ontology Language New Features and Rationale (Second Edition).* [Online] Available at: https://www.w3.org/TR/owl2-new-features/#F12:_Punning [Accessed 11.11.2025].

W3C, 2026. *W3C standards and drafts.* [Online] Available at: https://www.w3.org/TR/ [Accessed 16.2.2026].

Wilkins, D., 2023. *Implementing Full-Text Indexes.* [Online] Available at: https://www.dolthub.com/blog/2023-08-14-implementing-fulltext-indexes/ [Accessed 30.4.2026].

World Wide Web Consortium (W3C), 2015. *W3C Wiki GoodRelations.* [Online] Available at: https://www.w3.org/wiki/GoodRelations [Accessed 17.11.2025].

World Wide Web Consortium (W3C), 2022. *W3C Wiki Web Access Control.* [Online] Available at: https://www.w3.org/wiki/WebAccessControl [Accessed 17.11.2025].

Xu, A., 2025. *System Design Was HARD - Until You Knew the Trade-Offs.* [Online] Available at: https://www.youtube.com/watch?v=1nENigGr-a0&t=189s [Accessed 30.4.2026].

Zaidi, E. & Thoo, E., 2019. *Data Fabrics Add Augmented Intelligence to Modernize Your Data Integration,* Stamford, CT, USA: Gartner Group.

Zhuang, I. & Anderson, E., 2024. *Introduction to gRPC.* [Online] Available at: https://grpc.io/docs/what-is-grpc/introduction/ [Accessed 6.5.2026].

Index

A

ABox modelling 236–237
Abstract data structure 315
Academic exercise 478
Access control 461
Access control policies 459
Active Ontology tab 204
Actual value 120
Adult class 128
Agent 3
Agent pattern 208
Agile 165, 166, 223
  data management 4
  data process 134, 135
  development 20
AllegroGraph 59, 61, 81–83, 121, 154, 157, 246, 257, 298, 304, 436, 471
  database 2
  Gruff 418
  WebView GUI 434
Amazon Neptune 61, 246
Amazon Web Services 306
Anti-corruption layer 410
Anti-pattern analysis 161–162
Apache Airflow 136
Apache Jena 60, 61, 157, 305
ArangoDB 61
Artificial intelligence (AI) 1, 48
Associative relations 391
Auto-Detect value 83
Auto-generated ID 53, 54
Automated refactoring 285–287

Automated testing 289–291
Automatic testing tools 160
Automotive enterprise data lake
  architectural approach 459–460
  designs and manufactures 458–459
  domain semantics 461
  event-driven metadata 460–461
  implementation and scale 462
  PatientGraph semantic knowledge graph 464–468
  Semantic Knowledge Graph (SKG) 463–464
  semantic technologies 462
Avalara
  GraphRAG 469–470
  outcomes and impact 470
Axioms 202
  categories 108
  Open World Assumption (OWA) 110–112

B

Basic Formal Ontology (BFO) 202, 212
Basic Reusable Ontology (BRO) 100, 149–150
Big-data 343, 462
BigQuery 429
Blank node 119
Blazegraph 60, 61
Boundary Value Analysis (BVA) 225
Bounded contexts 406–413
Buckets 186–190

C

CamelCase 49, 50
Capturing provenance 256
Cardinality restrictions 108
Catalogs 415–416
C++ developer 91

Cellfie tool 155
ChatGPT 226
chatState parameter 436
Clarity 183
Class Description view 225
Classes 33
Class hierarchy 109, 189, 218
Class hierarchy tab 108, 110
Clinical trial matching 467–468
Closed World Assumption (CWA) 110, 260
Cloud landing zones 306–307
Coherence 183
Collection buckets 211
Collection pattern 211–212
Collection/role bucket 186
Commercial products 147
Commitment 183
Composability 353
Computational complexity problems 269–271
Concept of operations (ConOps) 165
Concept schemes 392
Conceptual modeling
  basics 181
  bottom-up design 220
  buckets 186–190
  class/instance 191–193
  classes and properties 182
  and design modeling 184–186
  design principles 183–184
  error detection 219
  large language models (LLM) 222–229
  multiple inheritance 190
  ontology 196–206
  ontology patterns 206–219

OWL model 182
type of property 193–195
visualization tools 181
WebProtégé 182
world *vs.* domain 191
Consistent semantic foundation 397
Constraints 121
Context maps 401–406
domain concept 408
relationship types 408
relationships 417
Context-specific values 437
Continuous integration and continuous deployment (CI/CD) 5, 134, 331
Continuous refactoring 142
Controlled vocabulary 372–373
Conventions 8–9
Conway's Law 354
Core technologies
axioms 107–112
Labelled Property Graphs (LPG) 58–65
RDF/RDFS semantic networks 47–58
SPARQL 65–86
Coupling 353
Cow class 109
Create discussion 11
Cross-context integration 401–406
customerID 341
Customer–supplier relationship 403, 410
Customization 192
Cypher 59

D

Darwin Information Typing Architecture (DITA) 470
Data catalog 46, 326, 413–415, 426

context-specific values 437
knowledge graph 433–436
large language models (LLM) 436–439
provenance 442–443
RAG system 432
search 435
user interface 437
Dataclass 311
Data fabrics 343, 351–354
Data governance 137
Data ingestion 154–155
data pipelines 250
Dehghani's Daff example 251
distinction 250
enterprises 250
graph separation 257
input data 254–255
knowledge graph 257
output files 257
overview 253–254
pattern 257–260
pipeline example 252–253
pipeline execution 255
placeholder values 250
StreamForge 251
Data integrity 113
Data lakes 46
Data life cycle 19
Agile 134, 138
building software 135
business domains 133
life-cycle activities 136–138
project 138–141
software development 135

Data Mesh 351–354
data pipelines 360
event and messaging backbone 355–363
interface definition 361
large language models (LLM) 361
orchestration, scaling, and discovery 359
relational data stores 357–358
semantic and vector data stores 359
service packaging and deployment 358
sidecars 361–363
Data pipelines 275
Data ports 413
Data product catalog
bounded contexts 425
canonical discovery operation 425
grounding semantic models 428–430
SPARQL query 427
Data products 350, 401, 409, 411–413, 415–416
data fabric 351–354
description 350
integration 351
operational and analytic concerns 350–351
port 334–335
reference architecture 355–363
Semantic Knowledge Graph (SKGs) 354–355
Data property assertions 189
Data quality levels 343
Dataset and catalog metadata 459
Data Catalog Vocabulary (DCAT) 431
Data transformations
data quality and validation 283
enterprise data 276
graph pipeline 276
record-level 278

relational (multi-dataset) transformations 279–280
relational database 276–278
semantic and business-logic 283
set-based 279
Datatype property
directional object properties 194–195
metadata 193–194
semantics/efficiency 196
subclass or collection/role 195–196
Debugging 313, 325
Decision-Makers 451–453
Defense Advanced Research Projects Agency (DARPA) 26, 44
Dehghani's Daff example 251
Description logic (DL) 32, 88, 95, 107
Description logic operators 108
Design constraints 293
CAP theorem 241–242
data transformations 275–283
sharding and graph databases 242–243
vertical and horizontal scaling 242
Design modeling 20, 136, 184–186, 240
architectural context 232
conceptual model 232
considerations 260–265
data ingestion 249–260
design constraints from distributed systems 240–244
GraphQL 293
implications 292
internationalization 293–294
language tags 294–295
minimal graph-theory primer 233–239
ontology 283–291
sub-graphs 244–249
Design ontologies 184

Design-time validation 288
Dichotomies 479
Directed acyclic graphs (DAGs) 234–236
Displayed inferences 16
DISTINCT keyword 266
Distributed governance 353
Distributed version control system (DVCS) 155
DL Query tab 14
docs_ontology 432
Domain 3
  concepts 406–410
  graph 255
  semantics 461
  teams 349
  vocabularies 200–201
Domain-driven design (DDD) 21, 194–195, 350, 396, 481
  and data products 398
  bounded contexts, ubiquitous language, and explicit models 398–416
  data catalog 396
  data product ports 421–424
  and data products 398
  domain model 396
  Semantic technologies 397
  StreamForge 416–420
Dublin Core 431
Dublin Core Metadata Initiative (DCMI) 39, 149, 197, 198

E

Elements 88–92
employee_id property 113
Encoding bias 183
End with 53
Enrichment 137, 142, 341, 344
Enterprise concerns 152

Enterprise data catalog project 165
Enterprise-grade triplestores 340
Enterprise-level video class 407
Enterprises 480
Epics 166
Error detection 219
Event-driven approach 355–363
Event-driven metadata 460–461
Evolution 411
Extendibility 183
Extract → Load → Transform (ELT) process 274–275
Extract → Transform → Load (ETL) process 274–275

F

Fifth Normal Form (5NF) 278
Fine-grained security 339
First-class architectural elements 424
First-class citizens 442–443
First-class concept 255
First-class objects 335
First Normal Form (1NF) 278
First-Order Logic (FOL) 95, 107
Free-Text Index (FTI) 445
Free trial software 151
Friend of a Friend (FOAF) 207
Functional properties 103

G

General Data Protection Regulation (GDPR) 172, 176, 336, 341
Generated SPARQL 313
GitHub 9, 11, 21, 30, 155–157, 160, 163, 421, 475
Globally unique 53
Global supply chains 471–474
Goldilocks zone 432
Google BigQuery 428

Governance 137, 142, 329, 330, 343
access interfaces 334
annotation 436–439
data asset 331
data product port 334–335
data products 334
domain-specific enhancement 341
and enrichment 335–336, 340–343
feedback 336–337
internal customers 338
managed assets 331
provenance 337–338
raw data 340
responsibilities and boundaries 331
security 338–340
Semantic Knowledge Graphs (SKG) 330
semantic policy models for governance (ODRL) 332–333
semantic systems 331
user-defined metadata 336–337
validation 341
Governance_Annotation instance 337
Graph-based reasoning 8
Graph database clients 41
Graph databases 158–159, 244, 302
GraphDB 59, 61, 83, 157, 246
Graphical user interfaces (GUIs) 4, 158, 308, 312
Graph Neural Networks (GNNs) 473
GraphQL 292, 293
Graph retrieval-augmented generation (GraphRAG) 469
GraphTalker 447–451
Graph-theory primer
graph-theoretic implications 235–237
graph-theoretic issues to tool functionality 237–239
network graph 234

OWL-based conceptual models 233
root nodes 234
Semantic Knowledge Graph (SKG) 239–240
visualization tools 238
Graph View 79
Graphwise 148, 281
Grounding semantic models 428–430
Grouping 279
Gruff visualization 2, 418, 420, 426

H

Hallucinations 7
Health Insurance Portability and Accountability Act (HIPAA) 341
Hierarchical relations 390
Horizontal scaling 242

I

Idempotence 312–313
Identity and access management (IAM) 306
Importing vocabularies 204–206
Individual Hierarchy Tab 376, 378
Individual requirements modeling
analyzing 174–175
appropriate requirements model 172–174
functional and non functional 167
stories 168–171
Information architecture (IA) 367
Information technology (IT) 5
Inner join 276
Instance 93
Interface Definition Languages (IDLs) 361, 421
Interface structure 423
Internationalization 293–294
Internationalized Resource Identifiers (IRIs) 44, 47–51
Internet 27

Interoperability 39
ISO 8601 197

J

JanusGraph 61
Java Virtual Machine (JVM) 302
JavaScript 302
Jay Gatsby 49
Jena 246

K

Kafka 428
Knowledge-Based Software Assistant (KBSA) 396
Knowledge graph 2, 325, 405, 406, 441
Knowledge graph programming 41–42
Knowledge graph visualization 154

L

Labelled property graphs (LPG) 29, 35–36, 45
  adoption levels 61
  knowledge graphs 58
  nodes and edges 58
  property graphs 59–61
  RDF-star 59–61
  Semantic Knowledge Graph (SKG) 62–63
Labels 55–56
Landing Zone 306, 307
Language tags 294–295
Large language models (LLMs) 1, 6, 162, 416, 466, 478
  conceptual entities 224–225
  data catalog knowledge 430
  data catalog knowledge graph 433–436
  generated test data 228–229
  ontologies and standards 430–433
  prompts 223

reasoner errors 229
StreamForge Data Catalog 430
subject matter 223–224
test data instances 225–228
vocabularies 431–432
Lexical vocabularies 373
Libraries 302
Life-cycle activities 136–138
Life-cycle phase 138
Linked data 36–37
notable linked data sources 37–39
programming languages 41–43
vocabularies 39–41
Lisp 302
Local systems 152
Logical inconsistencies 290
Logical operators 92–93
Low fidelity prototyping 177–178
Low-hanging fruit 139

M

Magic comment 78, 79
Managed services 152
Manchester Syntax 88
Mapping ports 422–423
Mapping relations 391
Market dominance 58
Measurement pattern 214–219
Membership 392
Meta graph 256
Metadata 6, 175–177, 193–194, 335–336, 352
Microservices 6, 349, 411
Microsoft Azure 306
Microsoft Docs
description 455

documentation 455
outcomes and impact 456–457
semantic approach 456
Multiple inheritance 91, 190

N

Named Entity Recognition (NER) 174
Named entity resolution (NER) 280–283
Namespaces 198–199
N-ary relations (property graphs) 209–211
Natural-language dialog 439–441
Neo4j 58, 203
NeOn tool kit 148
Neurosymbolic databases 315
New Entity Preferences 53
Node-level constraints 121
Non-Governmental Organizations (NGOs) 314
Notable linked data sources 37–39
Number of corpus documents 438
Numeric (Iterative) 53

O

Object-oriented patterns 206
Object-Oriented Programming (OOP) 91, 100, 138, 161, 370
Object property assertions 189, 218
Object Property Tab 390
OBO Foundry Ontologies 200
Online transaction processing (OLTP) 169, 277, 350
Ontologies 478
automated refactoring 285–287
automated testing 289–291
distributed systems 291–293
reusing 287
SPARQL testing 291
validation and quality checks 288–289

Ontology imports 204
Ontology modeling 148–149, 162
Ontology prefixes 204
OpenAPI 421
Open Digital Rights Language (ODRL) 431
Open-source libraries 147
Open-source solution 303
Open-source tools 152
Open World Assumption (OWA) 110–112, 260
Operational metadata 176
OPTIONAL anti-pattern 271–272
OPTIONAL clause 69, 71
OPTIONAL statement 77
Ordered value partition 212–214
OWLAPI 305
owl:differentFrom property 58
owl:equivalentClass 129
OWL Full 297
Owlready2 304, 305
owl:topDataProperty 72
owl:topObjectProperty 72

P

PatientGraph approach 466
Pattern matters 313
Pellet 16
People Ontology 49, 50, 56, 65, 82–83, 90, 93, 98, 110, 186, 215
Permission policy 336
Person class 98
Pizza tutorial 122
Playback Video Concept 408
Plugins 13
Polysemes 406–410
Ports 409
Product catalog 192

Production analytics platforms 428–430
Programming languages 158–159, 302
  applications 42–43
  graph database clients 41
  knowledge graph programming 41–42
Property assertions 218
Property-based restrictions 108
Property graphs 64
Property inferences 105
Property Knowledge Graphs 62–63
Property path 120
Protégé 50, 87, 109, 112, 115–119, 123, 125, 126, 148, 150, 151, 153, 157, 229
Prov W3C Working Group (PROV-O) 200, 431
Provenance 337–338, 442–443, 461
Provenance and lineage (PROV-O) 459
Public key infrastructure (PKI) 29
Published-language relationship 410
Punning 101
Python 301
  dictionary 312
  library 304–306, 437

Q

Quality metrics 338
Quantifier restrictions 108
QUDT ontology 214
Query-planner warnings 268

R

R 301
RDFLib 304
rdfs:comment 206
rdfs:label 31
Real-world use cases 29

Reasoner 16
Reasoner Active 17
Reasoner errors 229
Record-level transformations 278
Refactoring tools 160
Reflexive properties 104
Relational databases 242, 276–278
Relational data stores 357–358
Relational (multi-dataset) transformations 279–280
Remote procedure calls (RPC) 291
Rendering templates 311
Reposition 481
Repository control 83
Requirements acquisition, system narrative 165–167
Resource Description Framework (RDF) 19, 30
  International Resource Identifiers (IRIs) 48–51, 53–55
  labels 55–56
  prefixes 51–53
  resources 56–58
Resource Description Framework Schema (RDFS) 31, 47, 48
  See also Resource Description Framework (RDF)
REST-based ports 421–424
Retrieval-augmented generation (RAG) 7, 21, 140, 281, 314–315, 396, 433, 444
  data pipeline, vector for prompt 323
  generates response 324
  graph 326
  knowledge graph 325
  nearest neighbor vectors 323
  response and matching 324–325
Reusable vocabularies 149–150
  domain 200–201
  importing 204–206
  technical 197–200

upper model 201–204
ROBOT 284, 285, 291
Role class 213
Root-level problem 385–388
Row-level transformations 278
Rule Interchange Format (RIF) 28
Rust 301

S

Safety 312–313
Scala 302
Schema.org 77, 201
Security 329
Self-contained file 205
Self-service 352
Self-service access 354
Semantic design 274–275
Semantic Knowledge Graphs (SKG) 19, 20, 25, 33, 43, 46, 99, 114, 125, 138, 145, 231, 250, 291, 307, 322, 329, 333, 398, 414–416, 423, 426, 435, 445, 449, 452, 456, 457, 463–464, 472
accessing examples 10–12
advantages 63–64
architectural realities 240
architectures and systems 5–8
Avalara 469–471
context and problem 464
conventions 8–9
core capabilities and architecture 466–467
databases 150–153
description 1
design implications 243–244
design modeling 240
disadvantage 7
Essilor global supply chain 471–474
Graph Toolset 145–147

knowledge Graph 2
Labelled Property Graphs (LPGs) 62–63
layers 2
lessons 468
periodic predictions 239
self-contained resource 4
terminology 10
terminology heterogeneity 465–466
use case 467–468
users 169
Semantic metadata 175
Semantic models 353, 412
Semantic policy models for governance (ODRL) 332–333
Semantic tagging and controlled vocabularies (SKOS) 459
Semantic technologies 397, 400, 477–478, 481
data lake architectures 458–464
Microsoft Docs 455–457
Semantic Web 113, 196
community 54
conceptual and design models 31–34
defined 47
entities 33–34
foundation layers 30–31
labelled property graphs 35–36
linked data 36–37
modern stack 27–28
original vision 26–27
parameterized SPARQL functions 307–310
parameterizing SPARQL update 310–311
reasoners and query languages 34–35
vision *vs.* reality 28–29
Semantic Web Rule Language (SWRL) 34, 105–107
Semantic Web stack 150
Semantic Web Standards 36

Semiformal model 141
dimensions 368
higher-level languages 369
human and machine interpretation 369
organization chart 367
Simple Knowledge Organization System (SKOS) 370–375
synergy 368
Serializations 286
SERVICE keyword 76
Set-based transformations 279
Set theory 88–92
Severity 116
SHACL4Protege Constraint Validator 116
Shapes Constraint Language (SHACL) 19, 31, 261
blank node 119
data integrity 113
data integrity constraints 113
iterative methodologies 114
meta-model 118
Protégé 115–119
reporting *vs.* inconsistency 113
target class 118
validation 119–121
viewing graph 121–122
W3C standard 115
Sharding 242–243
Shared vocabulary 165
Shell-based program 284
Sidecars 361–363
Simple Knowledge Organization System (SKOS) 40, 45, 50, 55, 100, 149
axioms 389
controlled vocabulary 372–373
folksonomy 374
glossary 372

logical relations 388
metadata standards 373
methodological perspective 371
object properties 389–392
ontology development progresses 372
relationships 388
root-level problem 385–388
semantic relationships 371
standards-based way 370
taxonomy 373
taxonomy into ontology 381–385
thesaurus 374
to OWL ontology 375–380
Simplified Upper Level Ontology (SULO) 149, 150, 203, 431
Single Port 423
skos:narrower 386
Snap SPARQL query 93
SNOMED CT 201
Social metadata 176
Software development kits (SDKs) 303
Software engineers 370
Solution mappings 266
Source Shape 116
Space exploration 214
SPARQL 145, 397, 415, 425, 430, 434, 439, 447
SPARQL construct
cardinality expectations 263–264
datatype ranges 262–263
design implications 264–265
functional properties 263–264
SPARQL Engine 153
SPARQL functions 315
SPARQL queries 47, 51, 54, 67–70, 103, 237, 245, 289, 307–310, 324, 326, 379

AllegroGraph 83
anti-pattern 1 265–269
anti-pattern 2 271–272
anti-pattern 3 272–273
domain 90
heuristics 273–274
labels generation 70–73
linked data 73–80
operation clause 66
path operators 80, 85
permutations 265
property values 85
structure of 65–67
UNION 269–271
SPARQL-star 60
SPARQL templates 311
SPARQL UPDATE 310–311
Standardized modeling patterns 286
Standards-based models 64–65
Stardog 246
Stories
competency questions 171–172
oversized modeling story 170–171
requirements 168
size and modeling 168
SKG users 169
use cases 171
StreamForge 251
StreamForge Data Catalog 398, 406, 430
catalog-style metadata 417
conceptual product to implementation 419–420
context map relationships 417
core domain concepts 418
data products 418–419

first-class entities 417
natural-language dialog 439–441
OWL and SHACL 416
visualizing 418
streamforge_ontology 433
Structural checks 288
Subclass hierarchy 90
Subclasses 93
SubClassOf 3
Sub-graphs
capability 245
encapsulation boundaries 248
graph membership 245
language and client library 246
querying 249
standards 245–246
triples 247–248
Subject–predicate–object triples 31
Sub-property 104
SubPropertyOf 3
Super property 104
Supply chain risk management 473
context and problem 471–472
core capabilities 473
description 471
large-scale enterprise systems 474
Semantic Knowledge Graph (SKG) 472
use case 473
Symmetric properties 103
Synergy 368
System narrative
Agile 166
elements 166
enterprise data catalog project 165

individual requirements 167–175
low fidelity prototyping 177–178
metadata 175–177
Semantic Knowledge Graph (SKG) 165
shared vocabulary 165

T

Tagger 283
Tags 187, 188
Tag values 189
Taxonomy 381–385
Taxonomy-based reasoning 466
TBox 154, 235–236
Technical metadata 176
Technical vocabularies 197–198
Namespaces 198–199
standards documentation 199–200
Terminology 2, 10
Text strings 432
Textual annotations 443–444
TigerGraph 61
Timestamps 259
Tool-assisted approach 282
Tool functionality 237–239
TopQuadrant 59, 148
Trade-offs 314, 363
Transaction boundaries 158
Transformation rules 155
Transformations 275
Transitive properties 103
Tree-shaped structure 379
Trending record 259
Triple patterns 268
Triplestore 303–304, 339
Turtle 51, 52, 117, 125–126

TypeScript 302

U

Uniform Resource Identifiers (URIs) 30
Universally Unique Identifier (UUID) 53
User-defined metadata 336

V

Value restrictions 108
Vector embedding-based matching 466
Vendors 242
Venn diagrams 89, 90
Version control
  command-line expertise 156
  GitHub 156
  loading ontology files 156–158
  software and ontology 155
Vertical scaling 242
Very high-level languages (VHLL) 367
Video class 406
Video Playback 423
Video Playback Events 429
Visualization tools 237
Vocabularies 39–41, 149–150, 162, 196, 460
Vocabulary alignment 140
VocBench 148

W

W3C-Defined OWL Profiles
  design-time constraint 298–299
  OWL DL 296
  OWL EL 296
  OWL QL 296
  OWL RL 297
  product-Specific 297

W3C SKOS Working Group 431
W3C verifiable credentials 29
W3C Web Access Control (WAC) 40
Web Ontology Language (OWL) 2, 19, 31, 32
  annotation properties 100
  anonymous classes 96
  correcting errors 124
  data properties 124
  defined classes 96
  large-scale changes 124
  logic 92–93
  logical definitions 123
  object and datatype 102
  primitive classes 96
  property characteristics 102–104
  property inferences 105
  punning 101
  quantification operators 94–95
  relations and 97–99
  Semantic Web Rule Language (SWRL) 105–107
  set theory 88–92
  set theory with logic 93–94
  software 87
  system development methodology 86
  three kinds of properties 99–102
WebProtégé 148, 182, 220–222
WHERE clause 66, 67, 71, 122
Wide Web Consortiums (W3C) 26
Wikidata 73–80
WikidataQuery Service (WDQS) 66
Window 14, 15
Wizard for Documenting Ontologies (Widoco) 159

X, Y, Z

X.509 Certificates 29

Table 13-1 Example Context Map Relationship Types

Context map relationship	Meaning	How change propagates
Upstream Downstream	One bounded context produces data, events, APIs, or definitions that another bounded context consumes. The upstream context owns the source meaning; the downstream context may transform or reinterpret it for its own model.	Downstream contexts should perform impact analysis. A change in the upstream concept may affect downstream data products, analytics, recommendation logic, reports, features, or validation rules.
Customer Supplier	One bounded context supplies a data product, service, or model that another bounded context depends on. The supplier owns the product or model, but the customer has recognized requirements that should influence its evolution.	Changes may require negotiation, compatibility review, versioning, or updates to data contracts and SLAs. The supplier does not simply conform to the customer, but the customer's needs must be considered.
Published Language	A bounded context publishes a stable vocabulary, schema, ontology module, API, or data contract that other contexts can use for integration without adopting the publishing context's full internal model.	Internal changes do not automatically propagate. Other contexts are affected only when the published language or contract changes. Breaking changes should be versioned or introduced through a revised published interface.
Anti-Corruption Layer	A consuming bounded context protects its own model from another context's assumptions by translating, filtering, or reshaping incoming data or messages before they enter the local model.	The upstream change should not be absorbed directly. The translation layer must be reviewed to determine whether the change should be mapped, transformed, ignored, or isolated from the consuming context.
Conformist	A downstream bounded context intentionally adopts the upstream context's model rather than maintaining an independent translation or negotiation boundary. This can reduce integration cost but increases coupling.	Changes in the upstream model propagate more directly. The downstream context must usually adapt to the upstream change, which can simplify coordination but reduce local autonomy.

Table 8-2 Long/Normalized Table

Order_ID	Attribute	Value
O1	status	shipped
O1	total	120
O2	status	pending
O2	total	75

Table 6-1 Heuristics to Map Parts of Speech to Ontology Entities

Part of Speech or Phrase	Typical Ontological Representation	Example
Noun (Common, non-attributive)	Class or Instance	`Person, Vehicle`
Proper Noun (Proper Name)	Instance	`Paris, IBM, Alan_Turing`
Attributive Noun	Data property or sometimes Annotation property	`age, weight, population`
Adjective	Typically, Data Property values or Class constraints	`red, heavy, flammable`
Verb	Object Property	`locatedIn, manufactures`
Modal Verbs (can, must)	OWL Axioms, SHACL Constraints or Rules	`must_have_license`
Prepositions (in, on, of)	Usually represented as hierarchical relationships or specific Object Properties	`partOf, contains, managedBy`

Table 5-1 Basic Semantic Knowledge Graph Toolset

Kind of Tool	Purpose	Example
Ontology Modeling	Provides an environment to define classes, properties, and individuals; supports OWL syntax, reasoning, and visualization. Essential for creating the core conceptual and logical model.	Protégé
Reusable Vocabularies	Provide standard terms for metadata, labeling, and classification. Using them avoids reinventing the wheel and ensures compatibility with other datasets. Many vocabularies exist, but Dublin Core and SKOS are so widely adopted that entities from both appear in almost every ontology project.	Dublin Core, SKOS
Triplestore (Graph Database)	Stores RDF triples at scale; supports SPARQL querying, reasoning, indexing, and concurrent access. Needed when working with real datasets beyond what fits in memory. Commercial tools have many additional tools such as integration with LLMs, data ingestion, NLP analysis, and more.	AllegroGraph
SPARQL Engine	Executes SPARQL queries. Triplestores have one built in; for prototypes, tools like Gruff make it easy to run queries interactively. RDFLib also provides a lightweight engine.	Gruff or RDFLib SPARQL engine
Graph Visualization	Displays graphs for exploration and debugging; helps reveal structures and relationships that are hard to spot in raw triples.	Gruff
Data Ingestion	Instances can be created manually in Protégé. However, the process is arduous and won't support real data. There are various tools that can automate loading data from sources such as CSV files.	Cellfie
Database Client Library	Connects applications (Python, Java, etc.) to a triplestore via its API. Enables queries, inserts, and reasoning calls directly from code.	AllegroGraph Python Client

Table 8-3 Wide Form Table After Pivot

Order_ID	status	total
O1	shipped	120
O2	pending	75

Table 13-2 The Three StreamForge Meanings of the Video Polyseme

Shared term	Bounded context	Context-specific meaning
Video	Ingestion and Metadata	A managed media asset prepared for cataloging, storage, encoding, and metadata enrichment
Video	Playback and Engagement	A playable content item as experienced by a user session, including playback state, availability, and engagement events
Video	Content Discovery	A candidate item for ranking, personalization, similarity comparison, or recommendation generation

Table 5-3 Community Versions of Semantic Knowledge Graph Databases

Database	Free Version Name	License Type/ Access	Main Limitation	Local Install?
Amazon Neptune	AWS Free Tier	Managed cloud (AWS)	AWS account required;usagemay incur costs	✗
Stardog	Stardog Free/ Cloud Free	Proprietary(free license)	Community version only on cloud and limits to 1M edges and 3 databases	✗
GraphDB	GraphDB Free	Proprietary(free)	2 concurrent queries	✓
Apache Jena	Open Source	Apache2.0	None (Open Source)	✓
AllegroGraph	AllegroGraph Free Edition	Proprietary(free)	5Mtriples	✓

Table 13-4 Context Relations Between Bounded Contexts, Data Products, and Data Ports

From → To	Meaning	In OWL Terms
Domain Concept → Bounded Context	The meaning and validity of a domain concept are defined within a bounded context	object properties: in_bounded_context, SKOS semantic properties (broader, narrower, close match,…)
Data Product → Bounded Context	A data product is owned, governed, and semantically scoped by a bounded context	object property: has_bounded_context
Data Product → Data Product Port	A data product exposes interaction points for consuming or producing data	object property: has_port
Data Product Port → Port Implementation	A port is realized by one or more concrete technical implementations	object property: has_implementation
Bounded Context → Bounded Context	Contexts may depend on, translate, or protect against other contexts	Context Map + relationship type individuals

Table 8-1 Scaling and CAP Implications on Relational, Column, and Graph Databases

Model	Vertical Scaling	Horizontal Scaling	CAP Pressure
Relational	Excellent	Difficult but possible	Moderate
Column	Limited	Excellent	Low (workload tolerant)
Graph	Excellent	Hard and semantics-sensitive	High

Table 3-1 Semantic vs. Property Knowledge Graphs

Feature/Goal	Semantic Knowledge Graphs (OWL/RDF)	Property Graphs
Standards Compliance	Based on W3C standards (RDF, OWL, SPARQL)	RDF-star and SPARQL-star are making progress, but no standard with the traction of other Semantic Web technologies
Semantic Precision	Rich, formally defined semantics; supports reasoning, inference, and validation	Schemas are informal or optional
Properties on Properties (N-ary relations)	Not native but can be implemented with a common ontology design pattern	Native in the language
Identity & Linking	Uses IRIs for global identifiers; ideal for Linked Data, Data Mesh, and Data Catalogs	Typically uses local identifiers; linking across graphs is ad hoc
Extensibility & Reuse	Designed to reuse ontologies and vocabularies (e.g., DCMI, SKOS)	Reuse is manual; no native support for vocabulary alignment
Reasoning & Inference	Supports logical inference via OWL semantics; enables class hierarchy reasoning and constraint checking	Limited to custom logic; no native support for inference or constraints
Data Integration	Excellent for integrating heterogeneous, distributed data sources	Often requires custom ETL; lacks native support for schema alignment
Query Language	SPARQL: powerful and standardized, but can be less intuitive for path traversals	Cypher, GSQL, Gremlin, SPARQL-star: optimized for path queries and traversals
Graph Traversal Performance	Good but not as optimized for deep path traversal as Cypher	Excellent performance on traversals and graph algorithms
Big Data Ecosystem Integration	Emerging support (e.g., RDF4J + Hadoop); better suited to semantic data lakes than real-time pipelines	Strong integration with big data tools (Hadoop, Spark, etc.)
Use Cases	Knowledge management, semantic search, Data Mesh, Data Catalogs, compliance modeling, biomedical informatics	Algorithms designed for mathematical graph theory search, traversal, and computation

Table 8-4 Results of Running ROBOT on an Early Version of the People Ontology

Level	Rule Name	Subject	Property	Value
ERROR	duplicate_label	http://michaeldebellis.com/people/Max_Planck_Institute_for_Infection_Biology	rdfs:label	MaxPlanckInstitutefor Infection Biology
ERROR	duplicate_label	http://michaeldebellis.com/people/Max_Planck_Unit_for_the_Science_of_Pathogens	rdfs:label	MaxPlanckInstitutefor Infection Biology
ERROR	missing_label	http://michaeldebellis.com/people/Foot	rdfs:label	
ERROR	missing_label	http://michaeldebellis.com/people/canonical_unit	rdfs:label	
ERROR	missing_label	http://michaeldebellis.com/people/canonical_value	rdfs:label	
ERROR	missing_label	http://michaeldebellis.com/people/wikidata	rdfs:label	
ERROR	missing_ontology_description	http://michaeldebellis.com/people	dc:description	
ERROR	missing_ontology_title	http://michaeldebellis.com/people	dc:title	

Table 6-2 The Four Types of Metadata

Term	Description	Examples
Semantic Metadata	Conceptual modeling of the domain: classes, relationships, axioms, constraints. Often not considered "metadata" by ontologists.	OWL ontologies and domain vocabularies, FIBO, SNOMED, GoodRelations, gist
Operational Metadata	Provenance, authorship, versioning, access control.	Dublin Core, BIBO, PROV
Technical Metadata	Data types, formats, serialization rules, mapping to physical storage.	SHACL, XSD types, R2RML
Social Metadata	Ownership, stewardship, and accountability. Governance and compliance concerns, such as who owns which data and under what organizational conditions it can be shared.	PROV, DCAT, Open Digital Rights Language (ODRL)

Table 8-5 Standards for Integrating Distributed Systems

Pattern	Best Fit	Strengths	Weaknesses
REST (HTTP APIs)	General-purpose service integration	Simple, widely adopted, supported by all platforms; easy to test and debug	Can require multiple endpoints; may not align naturally with graph traversal
GraphQL	Client-driven queries over connected data	Allows clients to request exactly the data they need; well-suited for graph-shaped data; reduces over-fetching	Requires schema design and resolver logic
gRPC / RPC	Internal service-to-service	High performance; strongly typed interfaces; efficient serialization	Less human-readable; more complex tooling; not ideal for ad hoc exploration
Event / Message Bus (e.g., RabbitMQ, Kafka)	Asynchronous data exchange and streaming	Decouples producers and consumers; supports real-time pipelines and enrichment workflows; best for loose coupling and fault tolerance	Most complex; harder to trace and debug; requires additional infrastructure

Table 3-2 SPARQL Path Operators

Operator	Meaning	Example
/	Sequence (A followed by B)	wdt:P19/wdt:P131
|	Alternative (A *or* B)	wdt:P19|wdt:P131
*****	Zero or more repetitions	:has_social_relationship*
+	One or more repetitions	:contains+
^	Inverse (points the other way)	^rdfs:subClassOf (superclasses)
!	Negation of a property	!rdf:type (any prop but type)

Table 13-3 Video Concepts, Labels, and Bounded Contexts

Concept	Label used for tools	Preferred business term	Bounded context
`Video_Concept`	Video Concept	Video	None; cross-context organizing concept
`Ingestion_Video_Concept`	Ingestion Video Concept	Video	Ingestion and Metadata
`Playback_Video_Concept`	Playback Video Concept	Video	Playback and Engagement
`Recommendation_Video_Concept`	Recommendation Video Concept	Video	Content Discovery

Table 5-2 Additional Semantic Knowledge Graph Tools

Kind of Tool	Purpose	Example
Version Control & Collaboration Platform	Manages ontology files, source code, and documentation. Tracks history, supports team workflows, branching, merging, and working in local environments (e.g., the developer's hard drive) and synching local environment with hosted repository.	GitHub
Ontology/Graph Library	Works with RDF/OWL entirely in-process; good for prototyping, transformation, or systems that require a small footprint. Includes a SPARQL engine.	RDFLib
Ontology Documentation Generation	Automatically creates human-readable documentation from OWL ontologies. Reduces friction in sharing models with non-technical stakeholders.	Widoco
Automated Testing	Runs automated regression tests after significant changes to the system.	ROBOT
Refactoring Tools	Automate stereotypical changes to ontologies and knowledge graphs such as generating labels from IRIs, extracting parts of a knowledge graph, etc.	SPARQL query library
Anti-Pattern Analysis	Detects common ontology design mistakes and anti-patterns. Especially useful for beginners; guidance can be taken selectively.	OOPS!
Large Language Model (LLM)	Assists with drafting SPARQL, generating documentation, suggesting ontology terms and design, providing a basic pipeline for ingesting text and data,...	ChatGPT