188.399 Einführung in Semantic Systems

Author:

Max Tiessler

Collaborators:

Florian Haubenberger

2025-01-17

Contents

1 Introduction

This summary is for the course "Introduction to Semantic Systems" (course code: 188.399) at the Vienna University of Technology (TU Wien) and was created for the Winter Semester 2024 in preparation for the exam. The document covers most of the essential topics discussed in the course, focusing on key concepts and practical applications in semantic systems. While it aims to provide a comprehensive review of the material, some advanced topics might not be explored in full depth.

Almost all the content has been done following the subject's slides. Also personal examples and explanations from professional experience have been considered while doing this handbook.

This summary is available on GitHub. If you find typos, errors, want to add content (such as additional explanations, links, figures, or examples), or wish to contribute, you can do so at the following repository:
`https://github.com/mtiessler/188.399-Introduction-To-Semantic-Systems-Summary-TUW/blob/main/Course_Handbook.pdf`.

If the URL does not work, you can locate it on GitHub using the following details:

- **User**: mtiessler
- **Repo-Name**: 188.399-Introduction-To-Semantic-Systems-Summary-TUW

2   What is an Ontology

Ontology is derived from the Greek words *ontos* (being) and *logos* (word), meaning the study or systematic explanation of existence.

In philosophy, ontology is a branch concerned with categorizing and explaining existence. Aristotle (400 BC) made early attempts to establish universal categories for classifying everything that exists.

2.1   Definition

According to the **Merriam-Webster dictionary**:

- A branch of metaphysics concerned with the nature and relations of being.

- A particular theory about the nature of being and kinds of existents.

In the context of knowledge representation, **Studer (1998)** defines ontology as seen in figure 1. This definition highlights several key aspects:

- **Formal**: Ontologies are machine-readable and interpretable.

- **Explicit**: Concepts, properties, functions, and axioms are clearly defined.

- **Shared**: Ontologies are agreed upon and shared by a community.

- **Conceptualization**: They provide an abstract model of some phenomena in the world.
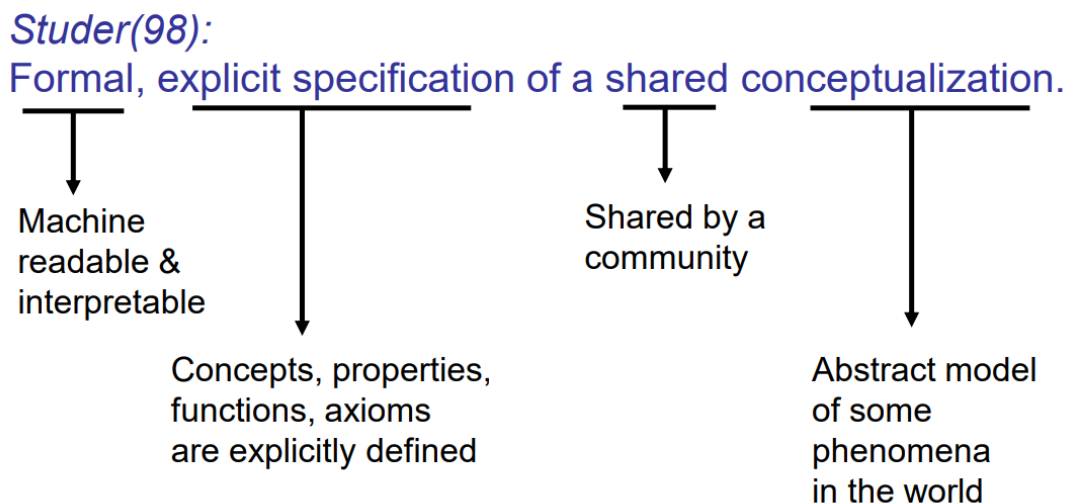
Figure 1: Ontology definition

2.2   Types and Categories
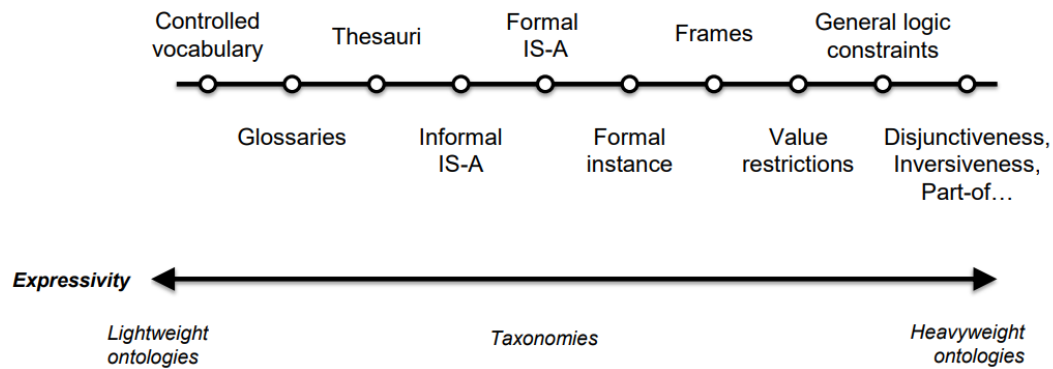
Figure 2: Ontology Types

2.2.1   Lightweight

Lightweight ontologies are simplified frameworks for organizing and defining knowledge. Key types include:

- **Controlled Vocabulary**: A finite list of terms, often used in catalogs.
  - **Example**: A library catalog listing book titles, authors, and genres (e.g., "Fiction," "Science," "Biography").

- **Glossary**: A controlled vocabulary with informal definitions provided in natural language.
  - **Example**: A glossary of medical terms, such as:
    * "Hypertension: High blood pressure."
    * "Cardiology: The branch of medicine dealing with the heart."

- **Thesauri**: A controlled vocabulary where concepts are connected through various relations:
  - **Equivalency**: Synonym relations between concepts.
    * **Example**: "Automobile" and "Car."
  - **Hierarchies**: Subclass and superclass relationships.
    * **Example**: "Dog" is a subclass of "Animal."
  - **Homographs**: Handling of homonyms (terms with identical spellings but different meanings).
    * **Example**: "Bank" (a financial institution) vs. "Bank" (the side of a river).
  - **Associations**: Relations between similar or related concepts.
    * **Example**: "Wine" is related to "Grape."

A popular example of a lightweight ontology is the "Friend of a Friend" (FOAF) ontology, which defines relationships between people in social networks.

2.2.2   Taxonomies

A taxonomy can be defined as: *a controlled vocabulary organized into a hierarchical structure.*

This can be seen in figure 3.

Taxonomies provide a hierarchical system of grouping concepts or entities. The term originates from the Greek words *taxis* (order, arrangement) and *nomos* (law, science). Types of taxonomies include:

- **Informal IS-A Hierarchy**: An explicit hierarchy of classes where subclass relations are not strict. For example, the index of a library, where a category like "Science" may include loosely related topics such as "Physics" and "Biology" without strictly defined boundaries.

- **Formal IS-A Hierarchy**: An explicit hierarchy of classes with strict subclass relations. For instance, in biology, "Mammal" is a strict subclass of "Animal," and "Dog" is a strict subclass of "Mammal."

- **Formal Instance**: A hierarchy that includes explicit class structures, strict subclass relations, and allows *instance-of* relations. For example, the classification of a specific entity like "Labrador Retriever" as an instance of the class "Dog," which is a subclass of "Mammal."
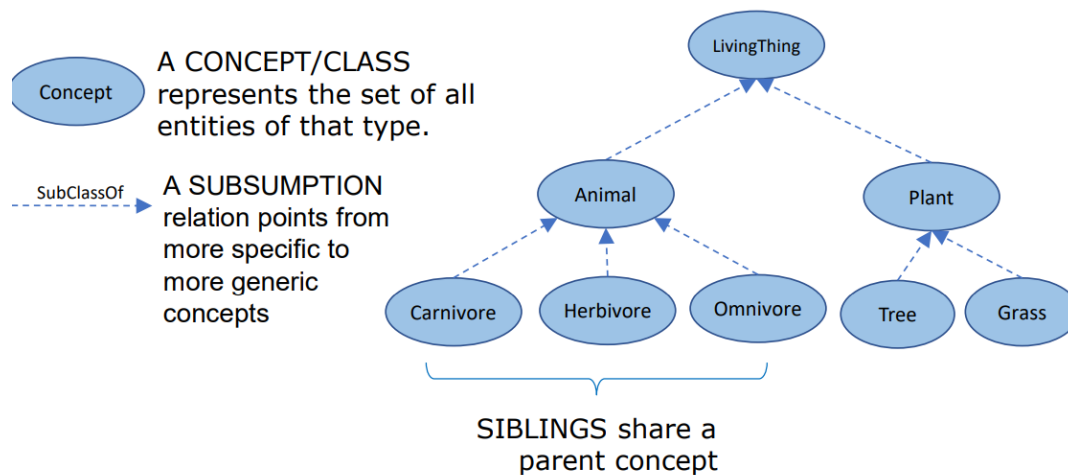
Figure 3: Taxonomy Example

Key concepts in formal taxonomies include:

- **Concept/Class**: Represents the set of all entities of a specific type.
- **Subsumption**: A specialization relation pointing from a specific concept to a generic one.

Subsumption/SubClass/Specialization Relation (in Formal IS-A Taxonomies)   Semantics (Meaning): Subsumption in formal IS-A taxonomies ensures that if one class is a subclass of another, all instances of the subclass are also instances of the superclass. For example, if *Herbivore* is a subclass of *Animal*, all instances of *Herbivore* are also instances of *Animal*.

Common Mistake: Subsumption is often mistakenly used to represent other types of relations, such as *part-of* (meronymy), which denotes a part-to-whole relationship rather than a class hierarchy.

2.2.3 Heavyweight

Heavyweight ontologies provide a rigorous framework for defining and organizing knowledge with a strong emphasis on logic and formal specifications. Key characteristics include:

- **Rigorous Definition and Organization of Concepts**: Concepts and their relationships are precisely defined, ensuring clarity and reducing ambiguity.

- **Focus on Logic**: Heavyweight ontologies prioritize formally correct deductions and inferences, enabling reliable reasoning and decision-making.

- **Careful Formal Specification**: A formal specification ensures consistency and eliminates contradictions within the ontology.

Heavyweight ontologies are often used in applications requiring advanced reasoning, such as artificial intelligence, semantic web technologies, and complex domain modeling. They aim to create a shared, consistent understanding of knowledge that can be processed by both humans and machines.

Examples of Heavyweight Ontologies

- **Frames**: Frames provide structured representations for defining classes and their properties.
  - **Example**: In a medical ontology, the frame for "Disease" might include properties such as "Symptoms," "Causes," and "Treatment."

- **Value Restrictions**: These enforce constraints on the values that properties can take.
  - **Example**: In a wine ontology, "Wine color" might be restricted to values like "Red," "White," or "Rosé."

- **General Logic Constraints**: These involve the application of logical rules to ensure consistency across the ontology.
  - **Example**: If "Animal" is defined as disjoint from "Plant," no instance can belong to both classes simultaneously.

- **Disjunctiveness**: Ensures that certain concepts are mutually exclusive.
  - **Example**: "Male" and "Female" in a gender ontology might be disjoint concepts.

- **Inversiveness**: Defines inverse relationships between properties.
  - **Example**: If "Parent" is a property, its inverse would be "Child."

- **Part-of Relationships**: These express compositional relationships.
  - **Example**: "Engine" is a part of "Car."

2.3 Ontology

Ontology is a **taxonomy** extended with additional relations and further constraints, providing a more comprehensive framework for modeling knowledge. Relations within an ontology are directed, pointing from a **Domain** concept to a **Range** concept. These relations can also include:

- **Inverse Relations**: Relations that allow bidirectional reasoning. For example, if "Tom eats Jerry," the inverse relation could be "Jerry is eaten by Tom."

- **Disjoint Concepts**: Concepts that describe non-overlapping instance sets, ensuring that instances belong to distinct categories. For example, *Carnivore* and *Herbivore* may be disjoint concepts.

2.3.1 Mechanics

The mechanics of an ontology focus on the logical and structural rules governing the relationships and organization of concepts, as seen in figure 4:

- Establishing the **Domain** and **Range** of relations to clarify which concepts are linked.

- Defining and maintaining **consistency** among concepts, relations, and instances.

- Implementing **constraints**, such as disjointness or cardinality, to refine the ontology's structure and enforce rules.
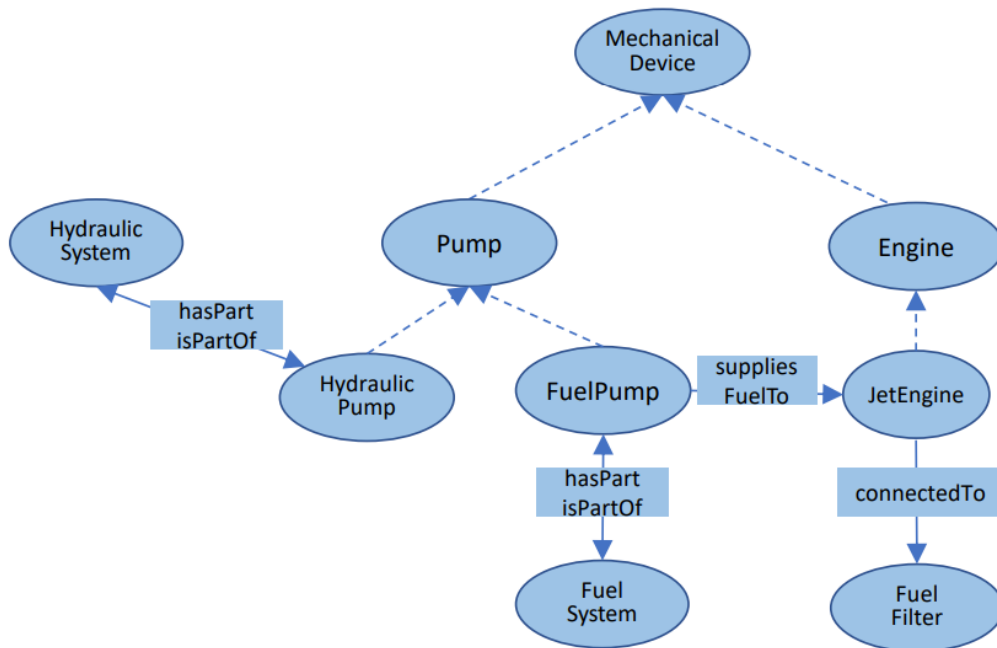
Figure 4: Ontology Mechanics

2.3.2 Instances/Entities

Instances or entities represent individual objects in the universe of discourse. They provide specific examples or metadata tied to the defined concepts in an ontology. This can be seen in 5.
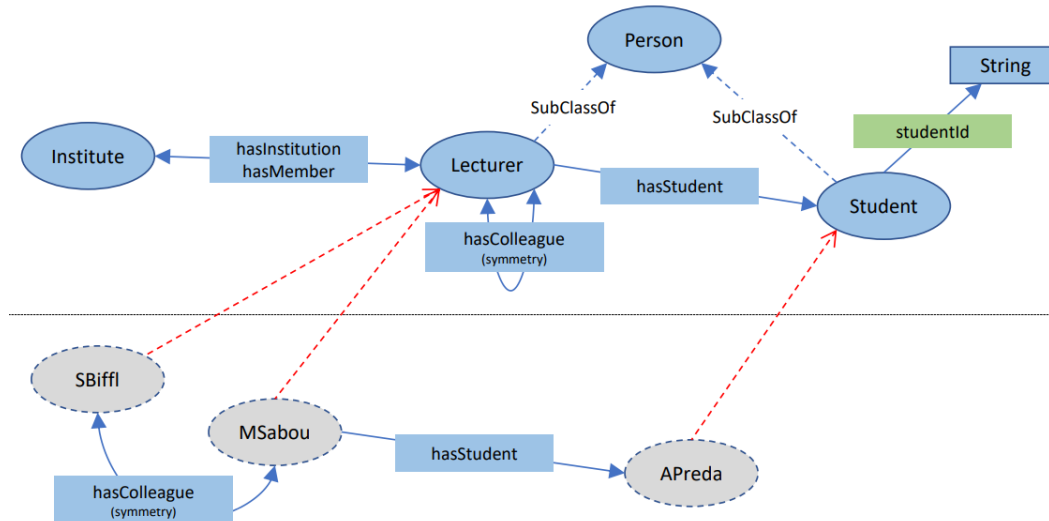
Figure 5: Ontology instances

Key aspects include:

- **Instance Representation**: An instance corresponds to a real-world entity and is associated with one or more concepts. For example:

  - *Tom is a Carnivore.*

  - *Jerry is an Animal.*

  - *Tom eats Jerry.*

- **Typing**: An instance is typed with respect to a concept, meaning that it is classified as being of a particular type. For example, "Tom" is an instance of the concept *Carnivore*.

- **Relations Between Instances**: Instances are connected through defined relations within the ontology, such as "eats" in the example above.

2.4   Description Logics

Description Logics (DL) are a family of formal knowledge representation languages used as the foundation for creating machine-readable ontologies. They are particularly suited for the Semantic Web and are based on a subset of First-Order Logics. DL provides a framework for representing and reasoning about knowledge in a structured and formal way.

2.4.1   Introduction

Description Logics aim to:

- Represent knowledge in a way that is both human-readable and machine-readable.

- Enable computers to reason with data and draw logical conclusions.

- Provide a formal basis for defining and organizing ontologies.

A formal, logic-based language in DL offers:

- **Syntax**: Defines which expressions are considered valid.

- **Semantics**: Provides the meaning of those expressions.

- **Calculus**: Defines how to compute and determine the meaning of expressions.

Semantic Web languages, such as OWL (Web Ontology Language), are built on Description Logics.

2.4.2 General DL Architecture

The general architecture of a Description Logic system (can be seen in 6), involves the following components:

- **Knowledge Base (KB)**: Composed of:

  - **TBox** (Terminological Knowledge): Contains definitions of concepts, attributes, and properties of a domain. For example:

    $$Writer \equiv Person \sqcap \exists author.Book$$

  - **ABox** (Assertional Knowledge): Contains information about specific individuals or entities. For example:

    $$Writer(GeorgeOrwell)$$

- **Inference Engine**: A component that reasons with the knowledge base to infer new information or check consistency.

- **Interface**: The mechanism for users or systems to interact with the ontology.
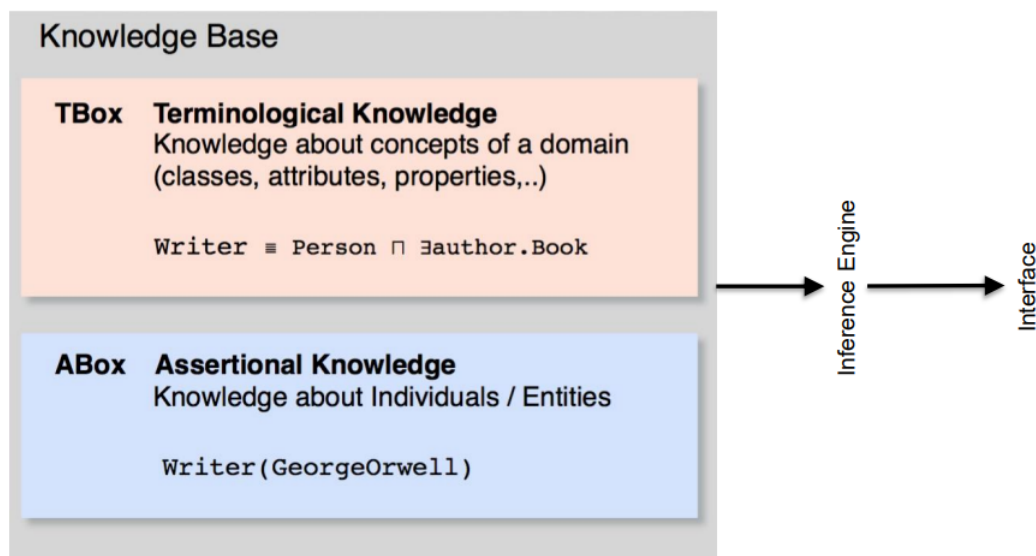
Figure 6: DL architecture

2.4.3   Notation / Syntax

The syntax of DL includes:

- **Atomic Types**:
    - Concept Names: $A, B, ...$
    - Special Concepts:
        * $\top$: Universal concept (all objects in the domain).
        * $\bot$: Bottom concept (empty set).
    - Role Names: $R, S, ...$
- **Constructors**:
    - Negation: $\neg C$
    - Conjunction: $C \sqcap D$
    - Disjunction: $C \sqcup D$
    - Existential Quantifier: $\exists R.C$
    - Universal Quantifier: $\forall R.C$

Examples:

- $Writer \equiv Person \sqcap \exists author.Book$
- $Novel \equiv Prose$
- $Novel \sqsubseteq Book$
- $Herbivore \sqsubseteq \neg \; Carnivore$
- $PetOwner \sqsubseteq Person \sqcap \exists hasPet.Animal$
- $Carnivore \sqsubseteq Animal \sqcap \forall eats.Animal$

2.4.4   Semantics

Description Logics rely on model-theoretic semantics, where an interpretation consists of:

- A **domain of discourse** (Δ): A collection of objects.
- **Interpretative Functions** (I): Functions mapping:
    - Classes (concepts) to sets of objects in the domain.
    - Properties (roles) to sets of pairs of objects.

In a DL, a class description characterizes the individuals that are members of that class, allowing precise reasoning and inference based on defined relationships and constraints.

It can be better seen in 7:
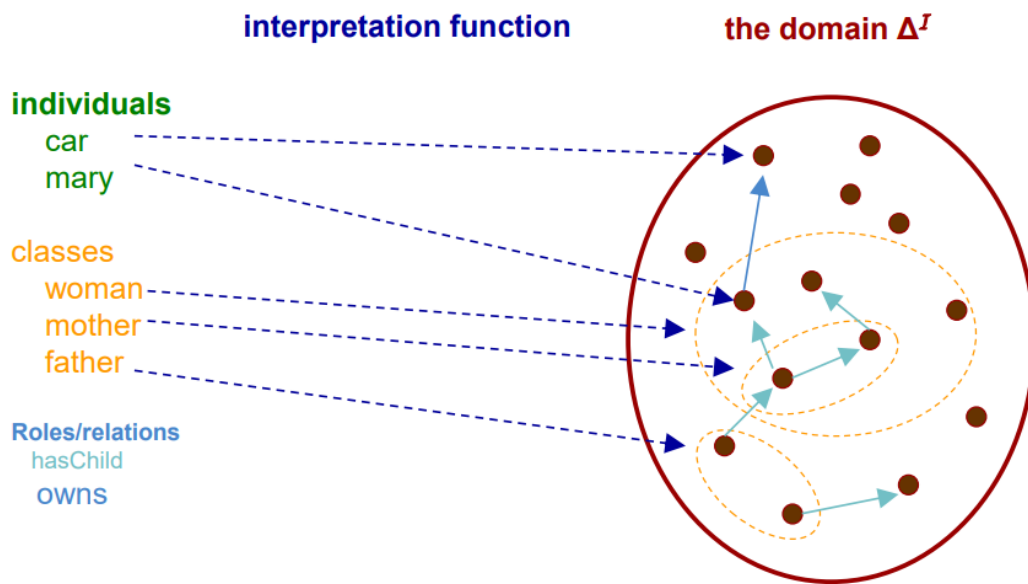
Figure 7: DL semantics

2.5 How to Build an Ontology

Building an ontology involves several steps, each addressing specific aspects of knowledge representation and organization. This process is iterative and is not strictly linear.
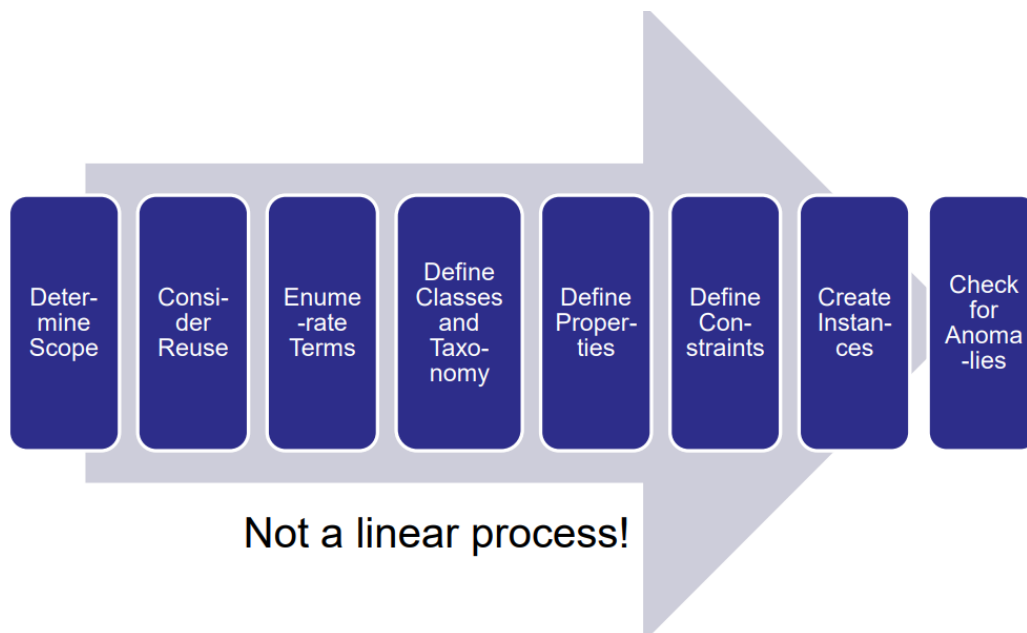
Figure 8: Ontology engineering process

2.5.1 1 - Determine Scope (Competency Questions)

The scope of the ontology should be determined by its intended use and anticipated extensions. Key questions to address include:

- What is the domain that the ontology will cover?

- For what purposes will the ontology be used?

- What types of questions should the ontology answer? (Competency Questions)

- Who will use and maintain the ontology?

Example Competency Questions for a Wine Recommender System:

- Which wine characteristics should I consider when choosing a wine?

- Is Bordeaux a red or white wine?

- Does Cabernet Sauvignon go well with seafood?

- What is the best choice of wine for grilled meat?

2.5.2　2 - Consider Reuse

Reusing existing ontologies saves effort, improves interoperability, and leverages validated ontologies. Sources for reusable ontologies include:

- Protégé Ontology Library (`http://protege.stanford.edu`)

- NCBO Bioportal (`http://bioontology.org`)

- Linked Open Vocabularies (`https://lov.linkeddata.es`)

2.5.3　3 - Enumerate Terms

Write down all relevant terms expected to appear in the ontology. This includes:

- **Nouns**: Basis for class names (e.g., wine, grape, winery).

- **Verbs or Verb Phrases**: Basis for property names (e.g., hasColor, madeFrom).

Examples for a Wine Ontology:

- Terms: wine, grape, winery, region, sugar content.

- Properties: hasColor, locatedIn, madeFrom.

2.5.4　4 - Define Classes and a Taxonomy

Classes represent concepts in the domain and are organized into a taxonomic hierarchy:

- A class is a collection of entities with similar properties (e.g., red wine, winery).

- Subclasses inherit properties from their superclasses.

Example:

- *Red Wine* is a subclass of *Wine*.

- Every instance of *Red Wine* is also an instance of *Wine*.

2.5.5   5 - Define Properties

Properties describe:

- **Attributes** of instances (e.g., color, sugar content).

- **Relationships** between classes (e.g., Wine *hasMaker* Winery).

Properties should:

- Be specified in the highest possible class in the hierarchy.

- Follow domain (parent class) and range (child class) rules.

2.5.6   6 - Define Constraints

Constraints refine the ontology by adding logical rules:

- **Cardinality:** Specifies the number of values a property can have (e.g., each wine has exactly one maker).

- **Existential Restrictions** ($\exists R.\,C$)

  - **Definition:** An existential restriction states that there is *at least one* relationship of type R to an individual belonging to class C. Formally:

  $$\exists R.\,C$$

  - **Interpretation:** "There is some R (relationship) to an individual in class C."

  - **Ensures that at least one property exists with a specified value.**

    * Example in general ontology terms: wines are located in regions, every employee works in at least one department, each student is enrolled in at least one course.

    * Symbolic example:
    $$\text{Parent} \equiv \exists\text{hasChild}.\,\text{Person}$$

    This means: "A *Parent* is something that has *at least one* 'hasChild' relationship to a *Person*."

- **Universal Restrictions** ($\forall R.\,C$)

  - **Definition:** A universal restriction states that *all* relationships of type R from an individual must go to members of class C. Formally:

  $$\forall R.\,C$$

  - **Interpretation:** "All R relationships go to individuals in class C."

  - **Ensures that all values of a property are of a certain type.**

    * Example in general ontology terms: wines can only be made in wineries, all employees must be part of the organization, all courses must belong to a recognized department.

* Symbolic example:

$$\text{Herbivore} \equiv \forall \text{eats. Plant}$$

This means: "A *Herbivore* is something that *only* eats *Plants* (i.e., every 'eats' relationship from a Herbivore goes to a Plant)."

- **Property Characteristics:** Includes logical characteristics of properties:

  - **Symmetry:** If $P(x, y)$ implies $P(y, x)$. *Example:* "marriedTo" implies if John is married to Mary, then Mary is married to John. Similarly, "friendOf" and "adjacentRegion" are symmetric properties.

  - **Transitivity:** If $P(x, y)$ and $P(y, z)$ imply $P(x, z)$. *Example:* "locatedIn" implies if City A is located in Region B, and Region B is located in Country C, then City A is located in Country C. Another example is "hasAncestor," where if John has an ancestor David, and David has an ancestor Robert, then John has an ancestor Robert.

  - **Inverse Properties:** If $P(x, y)$ implies $\text{InvP}(y, x)$. *Example:* "hasParent(John, Bob)" implies "hasChild(Bob, John)."
    Another example is "owns(Car, Owner)" implies "isOwnedBy(Owner, Car)."

Examples:

- Wine $\sqsubseteq \geq 1\text{madeFrom.Grape}$

- Wine $\sqsubseteq \leq 1\text{hasMaker}$

2.5.7   7 - Create Instances

Instances represent individual entities in the domain:

- The class becomes the direct type of the instance.

- Instances inherit properties and constraints from their class hierarchy.

- Property values assigned to instances must conform to constraints.

Example for a Wine Ontology:

- *Chateau Margaux* is an instance of *Wine.*

- *Chateau Margaux hasMaker Margaux Winery.*

2.5.8   8 - Check Anomalies

Use the formal semantics of the ontology to:

- Validate the model:

  - Check for consistency.

  - Ensure satisfiability of concepts and properties.

- Derive additional knowledge:

  - Automatically classify instances.

– Infer new relationships or properties.

2.6 Reasoning

Reasoning is one of the key functionalities provided by ontologies due to their formal grounding in logic. It enables the inference of new knowledge based on already declared information. Reasoning is performed using specialized software called reasoners or inference engines.

2.6.1 Capabilities of Reasoning with Ontologies

- **Consistency Checking**: Ensures that the ontology model is free from logical contradictions.
- **Satisfiability Checking**: Ensures that the ontology has no class or concepts that cannot have any instances.
- **Class Inference**: Discovers relationships between classes that were not explicitly declared.
- **Instance Inference**: Determines class membership of instances based on defined relationships and constraints.

2.6.2 Consistency Checking

Consistency checking ensures that the ontology does not contain logical errors:

- Example:
    - Declared: $Animal \sqsubseteq \neg Plant$
    - Declared: $Plant(myAlgae)$
    - Inferred: $Animal(myAlgae)$
    - Result: An inconsistency because myAlgae cannot be both an Animal and a Plant.

2.6.3 Satisfiability Checking

Satisfiability checking identifies concepts or classes that cannot possibly have any instances:

- Example:
    - Declared: $Algae \sqsubseteq Plant$, $Animal \sqsubseteq \neg Plant$, $Algae \sqsubseteq Animal$
    - Result: The class Algae is unsatisfiable because it cannot belong to both Animal and Plant simultaneously.

2.6.4 Class Inference

Class inference discovers new relationships among classes based on given knowledge:

- Example:
    - Declared: $Carnivore \sqsubseteq Animal \sqcap \forall eats.Animal$
    - Declared: $Cat \sqsubseteq Animal \sqcap \forall eats.Mouse$, $Mouse \sqsubseteq Animal$
    - Inferred: $Cat \sqsubseteq Carnivore$

2.6.5   Instance Inference

Instance inference identifies class membership for specific entities:

- Example:
  - Declared: hasPet(Person, Animal), PetOwner $\sqsubseteq$ Person $\sqcap$ $\exists$hasPet.Animal
  - Declared: Person(Pete), Animal(Spike), hasPet(Pete, Spike)
  - Inferred: PetOwner(Pete)

2.6.6   Class + Instance Inference

Combining class and instance inference allows the discovery of both new classes and instance memberships:

- Example:
  - Declared:
    - * hasPet(Person, Animal)
    - * PetOwner $\sqsubseteq$ Person $\sqcap$ $\exists$hasPet.Animal
    - * PetLover $\sqsubseteq$ Person$\sqcap$ > 3hasPet.Animal
  - Declared:
    - * Person(Pete), Animal(Tom), Animal(Jerry), Animal(Spike)
    - * hasPet(Pete, Spike), hasPet(Pete, Tom), hasPet(Pete, Jerry)
    - * $\{Spike\} \sqsubseteq \neg\{Tom, Jerry\}$
  - Inferred:
    - * PetOwner(Pete)
    - * PetLover(Pete)

3 Semantic Web, RDF, RDFS, and OWL

3.1 The Semantic Web

The Semantic Web represents an extension of the traditional Web, aiming to enable machines to understand and process data in a way that is closer to human reasoning. According to Tim Berners-Lee in the Scientific American journal:

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

In essence, it focuses on the meaningful interconnection of data to support better knowledge representation, information retrieval, and decision-making.

3.1.1 Key Concepts and Foundations

- **Linked Data:** The Semantic Web is built on the principle of Linked Data, which connects data across different resources using structured formats and qualified links.
  - Links between *things/entities* rather than just documents.
  - Knowledge is *machine-readable*, enabling automated reasoning.
  - Links are qualified to define relationships (e.g., `dbo:city` or `owl:sameAs`).
- **Comparison with the Web of Documents:**
  - The Web of Documents primarily links human-readable pages accessed via browsers.
  - The Web of Data (Semantic Web) links machine-readable entities accessed by programs and agents.
- **Scientific Perspectives:**
  - As emphasized by Sir Tim Berners-Lee: "The more things you have to connect together, the more powerful it is."
  - Decentralized databases underpin this ecosystem, supporting robust knowledge sharing and reasoning.

3.1.2 Semantic Web Language Design Principles

The design of the Semantic Web is governed by specific principles that enable flexibility and interoperability:

AAA Slogan On the Semantic Web, "Anyone can say Anything about Any topic." This principle highlights the openness of the web, allowing diverse sources to describe and link entities without centralized control. The Non-unique naming Assumption and the Open World Assumption stem from this slogan.

Non-unique Naming Assumption:

- The same entity may have multiple identifiers (URIs), reflecting its description by different sources.

- Explicit relationships such as `owl:sameAs` are necessary to state that two URIs refer to the same entity.

- Disjointness must also be specified explicitly when entities are distinct.

Open World Assumption (OWA):

- Missing information is not treated as negative information.

- The Semantic Web assumes incomplete knowledge, enabling reasoning over partial data.

- For example:

  - If `likes(PersonA, DrinkB)` is asserted, the OWA does not infer `not likes(PersonA, DrinkC)`.

  - Instead, it concludes `don't know` whether `PersonA` likes `DrinkC`.

- This contrasts with the Closed World Assumption (CWA), where missing information is treated as false.

3.1.3 Benefits of Semantic Web Principles

- Enables a flexible and decentralized approach to knowledge representation.

- Facilitates reasoning and integration of data from disparate sources.

- Supports robust applications in fields like knowledge graphs, data interoperability, and intelligent agents.

3.1.4 Technological Foundations

The Semantic Web leverages existing web technologies to achieve its goals. They are the bottom layer of the Semantic Web Technology stack, which can be seen in 9.

The technologies used are:

- **TCP/IP:** Facilitates data transfer across networks.

- **HTTP:** Enables client-server communication using a request/response model.

- **HTML and XML:**

  - HTML structures and displays web content.

  - XML provides a flexible markup framework for defining custom data formats.

- **URIs and IRIs:** Identifiers for resources on the web.

  - **URI Types:**

    * Uniform Resource Name (URN): Persistent, location-agnostic identifiers.
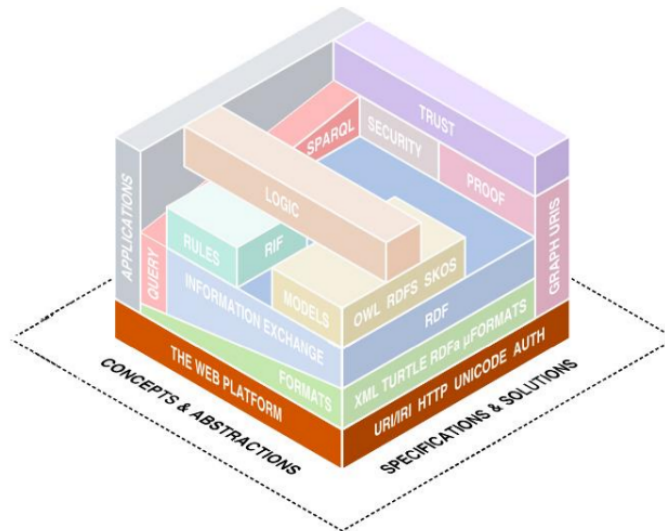
Figure 9: Semantic Web Stack

      ∗ Uniform Resource Locator (URL): Specifies access mechanisms for resources.

   – **IRI:** An internationalized extension of URI supporting Unicode characters.

3.1.5 Applications and Use Cases

- **Killer Applications:**
  - Intelligent agents retrieving, analyzing, and negotiating information.
  - Scheduling and coordination for multiple stakeholders (e.g., treatments or services).
- **Ontology and Knowledge Representation:**
  - Provides structured models to represent domains of knowledge.
  - Enhances the interoperability and reasoning capabilities of intelligent systems.

3.1.6 Challenges and Evolution

- The Semantic Web is still evolving, requiring:
  - Better adoption of Linked Data principles.
  - Tools for creating, managing, and consuming semantic data.
- Continues to build on the foundational technologies of the traditional web.

3.2 RDF

The Resource Description Framework (RDF) is a W3C standard designed to describe resources on the web in a structured and machine-readable way. It is foundational to the Semantic Web and provides a framework for knowledge representation and sharing across diverse systems.

3.2.1   Overview of RDF

RDF was developed in the late 1990s, with its final W3C recommendation (RDF 1.1) released in 2004. It uses a graph-based data model that formalizes semantics, making it accessible to machines. RDF expresses knowledge as a collection of statements, where each statement is represented as an RDF triple.

An RDF triple consists of:

- **Subject:** The resource being described, identified by a URI.

- **Predicate:** The relationship or property linking the subject to the object, also identified by a URI.

- **Object:** The value or related resource, which can be a URI or a literal.

3.2.2   RDF Graph Model

RDF statements naturally form a directed labeled graph, where:

- Nodes represent resources (subjects or objects).

- Arcs represent predicates, connecting subjects to objects.

For instance, the RDF triple:

```
<http://example.org/person/John>
<http://example.org/property/worksIn>
<http://example.org/resource/ProjectX>
```

is visualized as a graph:

$$\text{John} \xrightarrow{\text{worksIn}} \text{ProjectX}.$$

The object of one statement can act as the subject of another, enabling the integration of multiple graphs.

3.2.3   RDF Serialization Formats

RDF provides multiple serialization formats for different use cases, including machine communication, human readability, and ease of implementation. Common formats include RDF/XML, N-Triples, Turtle, and JSON-LD.

RDF/XML   RDF/XML is one of the earliest serialization formats, suitable for inter-machine communication.

- Useful for inter-machine communication.

- Every `Description` element describes a resource, referred to by an IRI.

- Every attribute or nested element inside a `Description` is a property of that resource.

An example:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://example.org/person/John">
    <worksIn rdf:resource="http://example.org/resource/ProjectX" />
  </rdf:Description>
</rdf:RDF>
```

N-Triples   N-Triples is a simple, line-based format where each RDF statement (triple) is terminated by a period.

- **Every statement is terminated with a full stop.**

- **URIs are enclosed in angle brackets (< >).**

- **Literals are enclosed in quotes.**

An example:

```
<http://example.org/person/John>
<http://example.org/property/worksIn>
<http://example.org/resource/ProjectX> .
```

Another example adapted from Euclid learning materials by Barry Norton:

```
<http://musicbrainz.org/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>
<http://xmlns.com/foaf/0.1/name>
"The Beatles".
<http://musicbrainz.org/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>
<http://www.w3.org/2002/07/owl#sameAs>
<http://dbpedia.org/resource/The_Beatles>.
```

In this example:

- The subject is `<http://musicbrainz.org/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>`.

- The predicate (property) is either `<http://xmlns.com/foaf/0.1/name>` or `<http://www.w3.org/2002/07/owl#sameAs>`.

- The object is `"The Beatles"` (a literal) or `<http://dbpedia.org/resource/The_Beatles>` (a URI).

N3 (Notation3)   N3 (Notation3) is a user-friendly RDF serialization format, designed to be easy to read and write.

- **Easy to read and write.**

- Syntax follows the structure: `<subject> <predicate> <object>`.

- **Every subject, predicate, and object is identified with a URI.**

- Two shortcuts for multiple statements about the same subject:

    - **;** introduces another predicate of the same subject.

    - **,** introduces another object with the same predicate and subject.

Example:

```
<#pat> <#child> <#al>, <#chaz>, <#mo>;
       <#age> 24;
       <#eyecolor> "blue".
```

Interpretation:

- Pat has the children Al, Chaz, and Mo.

- Pat's age is 24.

- Pat's eye color is blue.

Turtle   Turtle (Terse RDF Triple Language) is a human-readable serialization format and a subset of N3. It provides concise notations for repeated subjects and predicates, making it easier to read and write RDF data.

- **"Terse RDF Triple Language" (Turtle)** is a subset of N3.

- **Many URIs share the same basis**: Prefixes are used for namespace declarations.

- **Supports a shorthand for class membership**: The symbol a is used to indicate RDF type.

- **Simple syntax for data types and language tags for literals**.

- **Multiple statements about the same subject or predicate can be abbreviated**:

    - **;** introduces another predicate of the same subject.

    - **,** introduces another object with the same predicate and subject.

Example of prefixes and shorthand:

```
@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 mo: <http://purl.org/ontology/mo/>.
@prefix dbpedia: <http://dbpedia.org/resource/>.

@base <http://musicbrainz.org/>.

<artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>
    a mo:MusicGroup ;
    rdfs:label "The Beatles" ;
    owl:sameAs dbpedia:The_Beatles ,
<http://www.bbc.co.uk/music/artists/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#artist> .
```

Turtle also supports data types and language tags for literals. Example:

```
<recording/5098d0a8-d3c3-424e-9367-1f2610724410#_>
    a mo:Signal ;
    rdfs:label "All You Need Is Love" ;
    mo:duration "PT3M48S"^^xsd:duration .
```

```
dbpedia:The_Beatles dbpedia-owl:abstract
    "The Beatles were an English rock band formed (...) "@en ,
    "The Beatles waren eine britische Rockband in den (...) "@de .
```

Interpretation:

- **Class membership (a):** The subject `<artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>` is a member of the class `mo:MusicGroup`.

- **Labels (`rdfs:label`):** The label "The Beatles" describes the subject.

- **Literals with data types:** The duration `"PT3M48S"` is explicitly typed as `xsd:duration`.

- **Language tags:** Abstracts are tagged with language identifiers, e.g., `@en` (English) and `@de` (German).

3.2.4 RDF Literals and Datatypes

RDF supports two types of literals:

- **Plain Literals:** Text values with an optional language tag, e.g., `"Hello World"@en`.

- **Typed Literals:** Text values paired with a datatype URI, e.g., `"27"8sd:integer`.

The datatypes are based on XML Schema, defining value spaces, lexical spaces, and mappings between them.

3.2.5 RDF in Practice

RDF facilitates the integration of knowledge by enabling the merging of multiple graphs. For example, consider two datasets:

- Dataset A states `John worksIn ProjectX`.

- Dataset B states `ProjectX isIn NewYork`.

By combining these graphs, we infer that `John worksIn NewYork`, demonstrating RDF's power to create a web of interconnected knowledge.

3.3 Advanced RDF

Advanced RDF concepts extend the basic principles of RDF to accommodate more complex data structures, unnamed resources, and metadata about statements. These features support nuanced data modeling, enhance flexibility, and address specific challenges in RDF knowledge representation.

3.3.1 Structured Values

RDF enables the representation of structured values, such as addresses, by describing the "aggregated thing" (e.g., an address) as a separate resource. Statements about this new resource capture its individual components, such as street, city, state, and postal code.

Example in Triples Notation:

```
exstaff:85740 exterms:address exaddressid:85740 .
exaddressid:85740 exterms:street "Favoritenstrasse 9-11/188" .
exaddressid:85740 exterms:city "Vienna" .
exaddressid:85740 exterms:state "Vienna" .
exaddressid:85740 exterms:postalCode "1040" .
```

Example in Turtle:

```
exstaff:85740 exterms:address exaddressid:85740 .
exaddressid:85740 exterms:street "Favoritenstrasse 9-11/188" ;
                  exterms:city "Vienna" ;
                  exterms:state "Vienna" ;
                  exterms:postalCode "1040" .
```

3.3.2   Blank Nodes

Blank nodes are unnamed resources represented in RDF. They are useful for (1)**describing resources without assigning URIs**, (2)**grouping related information**, or (3)**representing n-ary relationships**. However, they pose challenges for reasoning and are discouraged in Linked Data, though they remain widely used.

Example Using Blank Nodes:

```
exstaff:85740 exterms:address _:johnaddress .
_:johnaddress exterms:street "1501 Grant Avenue" .
_:johnaddress exterms:city "Bedford" .
_:johnaddress exterms:state "Massachusetts" .
_:johnaddress exterms:postalCode "01730" .
```

Note: *Blank nodes (_) are not the same as variables (?) in SPARQL. Blank nodes represent unnamed resources in the data model, whereas SPARQL variables are placeholders used for querying and do not correspond to any specific resource in the RDF data. For example, _:johnaddress in RDF is a fixed blank node, while ?address in SPARQL is a query variable that can match any value in the data.*

3.3.3   Reification

Reification allows making statements about other statements, such as metadata, provenance, or trust. However, **it introduces complexity and potential for infinite recursion or cycles**.

Standard Reification Example:

```
:Bob :says _:s1 .
_:s1 rdf:type rdf:Statement .
_:s1 rdf:subject :Alice .
_:s1 rdf:predicate :likes .
_:s1 rdf:object :StarTrek .
```

Alternative approaches to reification include singleton properties, named graphs, and RDF-star (RDF*). Each method has trade-offs in terms of space efficiency, query performance, and system support.

Singleton Property Example:

```
:Alice :likes#1 :StarTrek .
:Bob :says :likes#1 .
:likes#1 rdf:singletonPropertyOf :likes .
```

Named Graph Example (TriG syntax):

```
:g1 {
  :Alice :likes :StarTrek .
}
:Bob :says :g1 .
```

RDF-star and RDF*   RDF-star introduces nested triples as first-class citizens, simplifying the representation of statements about statements. For example:

```
:Alice :says <<:Bob :likes :StarTrek>> .
```

```
<< :Alice :likes :StarTrek >> :assertedBy :Bob .
```

This approach is space-efficient and expressive but is not yet standardized across all triple stores.

3.3.4   RDF Data Structures: Containers and Collections

RDF supports two primary data structures:

- **Containers:** Open lists that allow the addition of new entries. Examples include `rdf:Bag`, `rdf:Seq`, and `rdf:Alt`.

- **Collections:** Closed lists where entries are fixed and cannot be extended.

Example of an RDF Collection in Turtle:

```
ex:fruits rdf:type rdf:List ;
          rdf:first "Apple" ;
          rdf:rest ( "Banana" "Cherry" rdf:nil ) .
```

3.3.5   RDF Vocabulary

The RDF vocabulary includes predefined classes and properties:

- **Classes:** `rdf:XMLLiteral`, `rdf:Property`, `rdf:Statement`, `rdf:Alt`, `rdf:Bag`, `rdf:Seq`, `rdf:List`.

- **Properties:** `rdf:type`, `rdf:first`, `rdf:rest`, `rdf:value`, `rdf:subject`, `rdf:predicate`, `rdf:object`.

3.3.6   Purpose of RDFS

RDFS serves two main purposes:

1. **Nomination:**

   - Defines the "types" (i.e., classes) of things about which we make assertions.

   - Specifies the properties that can act as predicates in these assertions to capture relationships.

2. **Inference:**

   - Allows reasoning over data to infer additional knowledge that is implicit based on the provided assertions.

3.3.7   Main Constructs of RDFS

RDFS introduces several key constructs to define classes, properties, and their relationships.

Classes:

- `rdfs:Resource`: The superclass of all resources in RDF.

- `rdfs:Class`: Declares a resource as a class (type or category) for other resources.

- `rdfs:Literal`: Represents literal values such as strings and numbers.

- `rdfs:Datatype`: A class of datatypes (e.g., `xsd:string`, `xsd:integer`), which is both an instance of and a subclass of `rdfs:Class`.

Properties:

- `rdfs:subClassOf`: Defines class hierarchies where properties of a superclass are inherited by its subclasses.

- `rdfs:subPropertyOf`: Defines property hierarchies where related resources by a subproperty are also related by its superproperty.

- `rdfs:domain`: Restricts the scope of a property to specific classes.

- `rdfs:range`: Restricts the range of values a property can have.

- `rdfs:comment`: Provides a human-readable description of a resource.

- `rdfs:seeAlso`: Indicates resources that provide additional information.

- `rdfs:isDefinedBy`: Points to a resource that defines the subject.

3.3.8   Example of Class Hierarchies

The following example demonstrates the use of `rdfs:subClassOf` to define hierarchies:

RDFS Example in Turtle:

```
@prefix ex: <http://example.org/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Person a rdfs:Class .
ex:Author rdfs:subClassOf ex:Person .
ex:Reader rdfs:subClassOf ex:Person .

ex:Frank a ex:Author .
ex:Lynda a ex:Reader .
ex:Frank ex:communicatesTo ex:Lynda .
```

From this data, an RDFS reasoner can infer:

- `ex:Frank` is an instance of `ex:Person`.

- `ex:Lynda` is an instance of `ex:Person`.

3.3.9   Domain and Range Restrictions

The `rdfs:domain` and `rdfs:range` properties restrict the applicability and type of properties.

Example in Turtle:

```
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

mo:member rdfs:domain mo:MusicGroup .
mo:member rdfs:range foaf:Agent .

mo:TheBeatles mo:member foaf:PaulMcCartney .
```

From this data, the following can be inferred:

- `mo:TheBeatles` is an instance of `mo:MusicGroup`.

- `foaf:PaulMcCartney` is an instance of `foaf:Agent`.

3.3.10   RDFS Vocabulary

The RDFS vocabulary expands RDF with the following classes and properties:

- **Classes:**
    - `rdfs:Resource, rdfs:Class, rdfs:Literal, rdfs:Datatype, rdfs:Container, rdfs:List.`

- **Properties:**
    - `rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range, rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy.`

3.3.11   Limitations of RDFS

Despite its utility, RDFS has significant limitations:

- It supports only basic inference and lacks the expressivity for complex reasoning.

- It does not allow for logical expressions beyond membership and basic hierarchies.

3.4   OWL (Web Ontology Language)

The Web Ontology Language (OWL) is a W3C recommendation designed for representing ontologies on the Semantic Web. OWL extends RDF and RDFS, providing a fully-fledged knowledge representation language that incorporates logical constructs and formal semantics. It enables more expressive and complex descriptions of knowledge domains while supporting inferencing to derive implicit knowledge.

3.4.1   Overview and Purpose

OWL addresses the limitations of RDFS by introducing a richer set of constructs and logical operators. Its formal foundation lies in Description Logics (DL), which provide:

- Well-defined semantics for precise reasoning.

- Mechanisms to verify the consistency of knowledge bases.

- Tools to infer implicit knowledge from explicitly defined statements.

- Computationally manageable reasoning processes.

OWL ontologies are typically expressed in RDF syntax, enabling easy integration and sharing on the Semantic Web.

3.4.2   Extending RDFS

OWL extends RDFS in several significant ways, including:

- Logical expressions such as `AND`, `OR`, `NOT`.

- Equality and inequality of classes and individuals.

- Local properties with domain- and range-specific constraints.

- Cardinality constraints, such as specifying the number of relationships.

- Symmetric, transitive, and inverse properties.

- Enumerated classes and required/optional properties.

For example, the following OWL statements specify the symmetry of a property:

```
:hasSibling a owl:ObjectProperty ;
          owl:SymmetricProperty .
```

3.4.3 Classes and Individuals

Classes in OWL define sets of individuals and are referred to as "types," "concepts," or "categories." Membership in a class is determined by logical descriptions rather than names. OWL defines two special classes:

- `owl:Thing`: The universal class that includes all individuals.

- `owl:Nothing`: The empty class containing no individuals.

Defining Classes and Individuals:

```
:Book a owl:Class .
:NineteenEightyFour a :Book .
:FrankSmith a owl:NamedIndividual .
```

3.4.4 Properties

OWL properties are binary relations between individuals or between individuals and data values. They come in two main types:

- **Object Properties:** Relate individuals to other individuals.

- **Datatype Properties:** Relate individuals to data values.

Examples of Object and Datatype Properties:

```
:author a owl:ObjectProperty ;
        rdfs:domain :Book ;
        rdfs:range :Writer .

:publicationYear a owl:DatatypeProperty ;
                rdfs:domain :Book ;
                rdfs:range xsd:integer .
```

Object properties can have characteristics such as transitivity, symmetry, and inverses:

```
:hasParent a owl:ObjectProperty ;
           owl:TransitiveProperty .
:hasBrother owl:SymmetricProperty .
:hasSpouse owl:inverseOf :isSpouseOf .
```

3.4.5 Class Hierarchies and Inference

OWL allows the creation of class hierarchies through subclass relationships, supporting reasoning to infer implicit relationships. For example:

```
:Novel a owl:Class ;
       rdfs:subClassOf :Book .

:Book a owl:Class ;
       rdfs:subClassOf :Work .
```

```
:Work a owl:Class .
```

From the above definitions, an OWL reasoner can infer that:

- :Novel is a subclass of :Work.

- Any individual of :Novel is also an individual of :Work.

OWL also supports the declaration of disjoint classes and equivalence:

```
:Book owl:disjointWith :Writer .
:Writer owl:equivalentClass :Author .
```

3.4.6   Advanced Constructs

OWL supports advanced constructs such as:

- **Cardinality:** Specifies the number of relationships an individual can have. For example:

```
:Person a owl:Class ;
        owl:equivalentClass [
            a owl:Restriction ;
            owl:onProperty :hasChild ;
            owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger ;
            owl:onClass :Child
        ] .
```

- **Enumerated Classes:** Defines classes by enumerating their members:

```
:PrimaryColors a owl:Class ;
               owl:oneOf (:Red :Blue :Yellow) .
```

- **Property Restrictions:** Enforces constraints on property values:

```
:Person a owl:Class ;
        owl:equivalentClass [
            a owl:Restriction ;
            owl:onProperty :hasAge ;
            owl:allValuesFrom xsd:integer
        ] .
```

3.4.7   Combining TBox and ABox

OWL integrates the TBox (terminology, or schema) and ABox (assertions, or instance data) into a unified framework. For instance:

```
:Book a owl:Class .
:Writer a owl:Class .
:NineteenEightyFour a :Book ;
```

```
                    :author :GeorgeOrwell ;
                    :publicationYear 1948 .
```

3.4.8   Inference and Reasoning

OWL reasoners leverage Description Logics to infer new knowledge from existing assertions. For example:

- Given `:Poet rdfs:subClassOf :Writer` and `:Writer owl:equivalentClass :Author`, a reasoner can infer `:Poet rdfs:subClassOf :Author`.

- Individuals can be related or distinguished:

```
        :NineteenEightyFour owl:sameAs :ARX012345 .
        :ARX012345 owl:differentFrom :ARX0123456 .
```

3.5   Advanced OWL

Advanced OWL concepts extend the foundational constructs to provide a more expressive framework for representing complex relationships, restrictions, and logical constructs in ontologies. These features enable nuanced reasoning and interoperability for knowledge representation on the Semantic Web.

3.5.1   Disjunctive Properties

Disjunctive properties ensure that two individuals cannot be related through both properties simultaneously. OWL provides constructs for declaring disjoint properties.

Example of Disjoint Properties:

```
:hasParent a owl:ObjectProperty ;
           owl:propertyDisjointWith :hasChild .
```

For multiple disjunctive properties, OWL provides a shortcut:

```
[] rdf:type owl:AllDisjointProperties ;
   owl:members (:hasParent :hasChild :hasGrandchild) .
```

3.5.2   Transitive Properties

A transitive property enables reasoning over chains of relationships. If a relationship exists between A and B, and between B and C, it can be inferred that the relationship exists between A and C.

Example of a Transitive Property:

```
:isPublishedBefore a owl:ObjectProperty ;
                   owl:TransitiveProperty ;
                   rdfs:domain owl:Book ;
                   rdfs:range owl:Book .
```

```
:AnimalFarm a owl:Book ;
           :isPublishedBefore :NineteenEightyFour .

:BraveNewWorld a owl:Book ;
              :isPublishedBefore :AnimalFarm .
```

From this, an OWL reasoner can infer:

```
:BraveNewWorld :isPublishedBefore :NineteenEightyFour .
```

3.5.3 Closed Classes (Nominals)

OWL allows the definition of closed classes using the `owl:oneOf` construct. This restricts membership in the class to explicitly enumerated individuals.

Example of Closed Classes:

```
:Novel a owl:Class ;
      owl:oneOf (:AnimalFarm :NineteenEightyFour) .

:AnimalFarm a :Novel .
:NineteenEightyFour a :Novel .
```

This states that only `:AnimalFarm` and `:NineteenEightyFour` are members of the `:Novel` class.

3.5.4 Property Restrictions

OWL supports property restrictions to define complex classes based on relationships. Restrictions include:

- `owl:hasValue`: Restricts a property to a fixed value.

- `owl:allValuesFrom`: Specifies that all values of a property must belong to a specific class (universal quantification).

- `owl:someValuesFrom`: Specifies that at least one value of a property must belong to a specific class (existential quantification).

- `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`: Restrict the number of relationships a property can have.

Example of Universal Restriction:

```
:VegetarianPizza a owl:Class ;
                rdfs:subClassOf [
                    a owl:Restriction ;
                    owl:onProperty :hasTopping ;
                    owl:allValuesFrom :VegetarianTopping
                ] .
```

Example of Existential Restriction:

```
:Reader a owl:Class ;
        rdfs:subClassOf [
            a owl:Restriction ;
            owl:onProperty :reads ;
            owl:someValuesFrom :Book
        ] .
```

Example of Cardinality Restriction:

```
:Tetralogy a owl:Class ;
           rdfs:subClassOf [
               a owl:Restriction ;
               owl:onProperty :hasVolumes ;
               owl:cardinality 4
           ] .
```

3.5.5   Property Characteristics and Relationships

OWL provides constructs for defining advanced property characteristics:

- `owl:SymmetricProperty`: A property is symmetric if the relationship is bidirectional (e.g., `isSiblingOf`).

- `owl:TransitiveProperty`: A property is transitive if it holds over chains of relationships (e.g., `isPartOf`).

- `owl:FunctionalProperty`: A property is functional if an individual can have at most one value for it (e.g., `hasBirthDate`).

- `owl:InverseFunctionalProperty`: A property is inverse functional if its inverse is functional (e.g., `isBirthMotherOf`).

- `owl:AsymmetricProperty`: A property is asymmetric if the inverse relationship cannot hold (e.g., `isLeftOf`).

- `owl:ReflexiveProperty`: A property is reflexive if it always relates an individual to itself (e.g., `isEqualTo`).

- `owl:IrreflexiveProperty`: A property is irreflexive if no individual can relate to itself (e.g., `isParentOf`).

Example of a Symmetric Property:

```
:isSiblingOf a owl:ObjectProperty ;
             owl:SymmetricProperty .
```

Example of an Inverse Property:

```
:hasSpouse a owl:ObjectProperty ;
           owl:inverseOf :isSpouseOf .
```

3.5.6   Combining Restrictions

OWL enables the combination of restrictions to define complex classes. For example:

```
:VegetarianPizza a owl:Class ;
            rdfs:subClassOf [
                a owl:Restriction ;
                owl:onProperty :hasTopping ;
                owl:someValuesFrom :VegetarianTopping
            ] ;
            rdfs:subClassOf [
                a owl:Restriction ;
                owl:onProperty :hasTopping ;
                owl:allValuesFrom :VegetarianTopping
            ] .
```

This defines :VegetarianPizza as a class where all toppings are vegetarian and at least one topping is specified.

3.6   RDF, RDFS, OWL Summary

The following table summarizes the key features of RDF, RDFS, and OWL, highlighting their roles, capabilities, and how they extend one another:

Aspect	RDF	RDFS	OWL
Purpose	Basic framework for representing data on the web in a graph-based model.	Extends RDF by providing vocabulary for schema definitions and lightweight reasoning.	Extends RDFS with advanced ontology capabilities, formal semantics, and logical reasoning.
Key Features	Uses triples: `(subject, predicate, object)`. Basic graph-based data representation. Supports linking resources using URIs. Serialization formats like RDF/XML, Turtle, JSON-LD.	Adds classes (`rdfs:Class`) and properties (`rdfs:Property`). Defines hierarchies: `rdfs:subClassOf`, `rdfs:subPropertyOf`. Allows specifying `rdfs:domain` and `rdfs:range`. Supports basic inference over data.	Supports advanced reasoning using Description Logics. Adds property characteristics: symmetry, transitivity, inverses. Allows logical expressions: `AND`, `OR`, `NOT`. Defines cardinality restrictions (`min`, `max`). Supports enumeration and disjointness of classes.
Use Cases	Simple data linking and integration. Sharing machine-readable data. Foundational layer for semantic technologies.	Lightweight schema definitions. Organizing data into class and property hierarchies. Enabling limited inferencing.	Building complex ontologies for knowledge representation. Supporting automated reasoning and decision-making. Applications in AI, knowledge graphs, and intelligent agents.
Extension	Foundational technology.	Built on RDF; extends it with schema capabilities.	Built on RDFS; adds expressive ontology language and formal semantics.
Limitations	No inferencing capabilities. Limited expressiveness for representing schemas.	Limited support for complex reasoning. Cannot express advanced logical constraints.	Higher complexity for reasoning. Increased computational overhead.

Table 1: Summary of RDF, RDFS, and OWL

4 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is the W3C-standardized query language for RDF (*Resource Description Framework*). It enables applications to retrieve, manipulate, and update RDF data by expressing queries that match graph patterns. SPARQL queries operate over one or multiple RDF graphs, which are essentially sets of subject–predicate–object triples. In the sections below, we will explore the core ideas of SPARQL, from basic queries and solution modifiers to advanced features like optional patterns, federated queries, and SPARQL 1.1 updates.

4.0.1 Basic Graph Patterns

The core mechanism of a SPARQL query is graph pattern matching. A set of triple patterns, enclosed in braces {}, specifies the structure that needs to be matched in the data. Each triple pattern can contain **variables** (denoted with a question mark, such as `?album` or `?track`), **IRIs**, or **literals**. SPARQL queries have a *standard structure* composed of several parts:

1. **Prefix Declarations:**
   These introduce shorthand aliases (`PREFIX dbpedia: <...>`) so that IRIs can be written in a shorter form.

2. **Query Form:**
   Determines the shape of the query's output. Common forms include:

   - `SELECT` – Retrieves variables bound by the query.

   - `CONSTRUCT` – Builds an RDF graph from matched bindings.

   - `ASK` – Returns a boolean indicating whether any solution exists.

   - `DESCRIBE` – Returns RDF data about a resource, according to the endpoint's discretion.

3. **WHERE Clause:**
   Contains one or more **basic graph patterns** (BGPs), typically enclosed in curly braces {}. These patterns specify the triples that must match, using **variables**, **IRIs**, and **literals**.

4. **Solution Modifiers:**
   Optional clauses that refine the result set (e.g., `ORDER BY`, `LIMIT`, `OFFSET`, `GROUP BY`, `HAVING`, etc.). These can reorder, trim, or aggregate the solutions.

4.0.2 SPARQL Query Forms

SPARQL supports four primary query forms:

1. `SELECT` query returns variable bindings in a tabular structure, serving as the most common way to retrieve data

2. `ASK` is used to check for the existence of a certain pattern, it returns a simple boolean `true` or `false`

3. `DESCRIBE` For exploration of unfamiliar data. Returns additional statements considered relevant by the SPARQL endpoint about resources matching the query.

4. `CONSTRUCT` queries transform the original RDF graph into a new RDF graph. They allow you to specify a *template* in which matching variables are substituted with actual data, thus producing new triples. This is especially useful for reformatting data from one vocabulary to another or for materializing inferred statements for later use.

4.0.3   Solution Modifiers: Filtering, Ordering, and Aggregation

After extracting candidate solutions, it is often necessary to refine them. SPARQL *solution modifiers* accomplish this in various ways. Notable solution modifiers include:

1. `FILTER`: Restricts solutions by applying boolean conditions (e.g., numeric comparisons, string patterns, language tags).

2. `ORDER BY`: Reorders query results based on specified variables, ascending or descending.

3. `LIMIT` **and** `OFFSET`: Allows you to retrieve only a subset of the results, for example, the first 10 or those starting from a particular index.

4. `COUNT, SUM, AVG, MIN, MAX`: Aggregates that can be used to compute statistics over groups of results.

5. `GROUP BY`: Organizes results into groups based on one or more variables, enabling the use of aggregate functions.

6. `HAVING`: Filters groups (created by `GROUP BY`) according to aggregate conditions (e.g., `HAVING(SUM(?duration) > 3600)`).

Example:

```
PREFIX mo:  <http://purl.org/ontology/mo/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?album (SUM(?duration) AS ?totalDuration)
WHERE {
    ?track a mo:Track ;
           mo:track_duration ?duration ;
           mo:track_album ?album .
}
GROUP BY ?album
HAVING (SUM(?duration) > 3600)
ORDER BY DESC(?totalDuration)
LIMIT 10
```

In this single query:

- `GROUP BY ?album` groups all tracks belonging to the same album.

- `SUM(?duration)` computes the total playtime per album.

- `HAVING (SUM(?duration) > 3600)` discards any album whose total duration is shorter than 3600 seconds.

- `ORDER BY DESC(?totalDuration)` sorts the remaining albums from the longest total duration to the shortest.

- `LIMIT 10` returns only the first ten results in that sorted list.

4.0.4   Advanced Query Patterns: UNION, OPTIONAL, and Subqueries

Beyond basic patterns, SPARQL includes more advanced constructs to handle complex retrieval scenarios:

1. `UNION`: Performs a logical disjunction by matching data that satisfies one pattern or another (or both). This is useful for gathering data from multiple potential paths or classes in a graph.

2. `OPTIONAL`: Retrieves data only if it exists. If the pattern does not match, the solution remains valid, but the variable bound to the `OPTIONAL` pattern remains unbound. This is especially helpful in heterogeneous datasets where certain properties may be missing.

3. `Subqueries`: Allows you to embed a complete `SELECT`, `CONSTRUCT`, or other query form inside a larger query. This technique lets you compute intermediate results (including aggregations) and then utilize them in the outer query for more sophisticated logic.

Below is a *brief complex example* that demonstrates using `UNION`, `OPTIONAL`, and a subquery in the same statement. In this scenario, we assume a dataset describing employees and researchers, some of whom have co-authors, supervisors, and birthdates:

```
PREFIX ex: <http://example.org/schema#>
PREFIX schema: <http://schema.org/>

SELECT ?person ?boss ?coAuthor ?birthDate ?coAuthorCount
WHERE {
  # Combine employee/boss relationships with researcher/coAuthor relationships:
  {
    ?person a ex:Employee ;
            ex:hasBoss ?boss .
  }
  UNION
  {
    ?person a ex:Researcher ;
            ex:hasCoAuthor ?coAuthor .
  }

  # Retrieve optional birthdate information:
  OPTIONAL {
    ?person schema:birthDate ?birthDate .
  }

  # Subquery: count the total number of coAuthors per person:
  {
    SELECT ?person (COUNT(?coAuthor) AS ?coAuthorCount)
    WHERE {
```

```
      ?person ex:hasCoAuthor ?coAuthor .
    }
    GROUP BY ?person
  }
}
```

In this example:

- **UNION** joins two distinct patterns: one matching employees and their bosses, and the other matching researchers and their co-authors.

- **OPTIONAL** captures the person's **birthDate** if it exists; otherwise, that variable remains unbound.

- A **Subquery** computes **?coAuthorCount** by aggregating all co-authorships for each person. The result of this subquery is then available to the outer query to be combined with boss and birthdate information.

4.0.5   Federated Queries

SPARQL also supports the notion of federated queries, allowing you to query multiple endpoints in a single statement. By using the **SERVICE** keyword, you can direct certain parts of your query to remote SPARQL endpoints. This enables data retrieval from diverse sources across the web of Linked Data, seamlessly integrating them into one coherent result set. A typical use case might involve fetching information about movies from one endpoint and cross-referencing it with actor data from another.

4.0.6   Entailment Regimes

One of the key advantages of RDF is its schema and inference capabilities. In SPARQL, reasoners can implement various *entailment regimes* (e.g., RDFS or OWL-based reasoning) to infer new facts beyond those explicitly stated. For example, if `mo:MusicGroup` is declared a subclass of `mo:MusicArtist`, then an individual typed as a `mo:MusicGroup` will also be considered a `mo:MusicArtist` in queries, provided the store or endpoint supports reasoning. These inferred facts can significantly enrich query results and reveal deeper relationships in the data.

Below are two common RDF entailment patterns that illustrate how reasoning can affect query results:

1. `rdfs:subClassOf` Reasoning

- **Pattern Description:** In RDFS (RDF Schema), if we declare that one class is a subclass of another, any individual typed as the subclass also implicitly belongs to the superclass.

- **Example Setup:**

```
@prefix :    <http://example.org/> .
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:Beatles a mo:MusicGroup .
```

```
# Declaration: mo:MusicGroup is a subclass of mo:MusicArtist
mo:MusicGroup rdfs:subClassOf mo:MusicArtist .
```

- **Query Comparing No Reasoning vs. With Reasoning:**

```
SELECT ?artist
WHERE {
  ?artist a mo:MusicArtist .
}
```

- **Without Reasoning:**

  – The triple :Beatles a mo:MusicGroup does not directly assert
    :Beatles a mo:MusicArtist.

  – Therefore, this query returns *no results*, as :Beatles is strictly typed as mo:MusicGroup.

- **With RDFS Reasoning:**

  – The reasoner infers :Beatles a mo:MusicArtist, because
    mo:MusicGroup is a rdfs:subClassOf mo:MusicArtist.

  – This query now returns :Beatles (among other mo:MusicArtist individuals).

2. **rdfs:domain Reasoning**

   - **Pattern Description:** If a property P has an rdfs:domain of C, then whenever we see a triple
     (X P Y), we can infer that X must be an instance of C.

   - **Example Setup:**

```
@prefix :   <http://example.org/> .
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .


:beatles mo:hasBandMember :john .


# Declaration: the domain of mo:hasBandMember is mo:MusicGroup
mo:hasBandMember rdfs:domain mo:MusicGroup .
```

   - **Query Comparing No Reasoning vs. With Reasoning:**

```
SELECT ?group
WHERE {
  ?group a mo:MusicGroup .
}
```

   - **Without Reasoning:**

     – :beatles has not been explicitly typed as a mo:MusicGroup.

     – The above query yields *no results*, since there's no direct triple
       (:beatles a mo:MusicGroup).

   - **With RDFS Reasoning:**

- – From the triple `:beatles mo:hasBandMember :john`, plus the domain declaration
  `mo:hasBandMember rdfs:domain mo:MusicGroup`, the reasoner infers
  `:beatles a mo:MusicGroup`.

- – Now the query returns `:beatles`, because the reasoning engine infers the missing type
  information.

4.0.7   SPARQL 1.1 Updates

Initially, SPARQL 1.0 provided powerful querying features but no direct mechanism to change RDF data. SPARQL 1.1 introduced a suite of update operations. `INSERT DATA` and `DELETE DATA` enable adding or removing specified triples, while the combined `DELETE/INSERT` pattern is ideal for modifying existing data in a single step. These updates typically appear in a `WHERE` block that selects which data to remove and how to transform it for insertion.

Additionally, SPARQL 1.1 offers graph-level operations such as `CREATE`, `CLEAR`, and `DROP` to manage named graphs in a triple store, as well as `LOAD` to import data from external files. Finally, operations such as `MOVE`, `COPY`, and `ADD` facilitate graph-level data management by transferring triples between named graphs.

5 SHACL

The Shapes Constraint Language (SHACL) is a W3C standard[1] designed for validating and describing constraints on RDF graphs. These constraints capture how data should be structured and related, thereby increasing trust in the quality and consistency of the underlying information. SHACL provides a rich vocabulary for defining and organizing shapes, which can specify constraints at the node level and at the property level. When a dataset is validated against these shapes, a validation report is produced, indicating any violations or potential issues.

5.0.1 Overview of SHACL

At its core, SHACL consists of two types of RDF graphs: a *data graph* and a *shapes graph*. The data graph contains the actual assertions about entities and their relationships, while the shapes graph describes how these entities are expected to appear. During validation, nodes in the data graph are checked against the shapes in the shapes graph, and any constraint violations are recorded in the final validation report.

The shapes graph itself is composed of one or more shapes. These shapes can specify conditions such as the number of allowed values for a given property, the type of those values, or logical combinations of constraints. Since shapes and constraints are expressed in RDF, the same data modeling environment used for domain data can also describe the validation rules.

5.0.2 Targets and Focus Nodes

Central to the SHACL validation process is the concept of *targets*, which determine which nodes in the data graph need to satisfy a given shape. These targeted nodes are called *focus nodes*. SHACL provides various kinds of targets. In practice, one might specify a target class so that all individuals of that class are validated against the shape, or even select a specific node directly. Other target types include all subjects of a certain property or all objects of a certain property. Once the SHACL processor identifies the focus nodes from the targets, it applies the relevant constraints to those nodes and checks for compliance.

Example. Suppose we have a shape named `ex:PersonShape` that is intended for individuals of type `ex:Person`. By declaring

```
ex:PersonShape
  a sh:NodeShape ;
  sh:targetClass ex:Person .
```

all nodes in the data graph that are of type `ex:Person` become focus nodes and must fulfill the shape's constraints.

5.0.3 Node Shapes and Property Shapes

SHACL distinguishes between two general categories of shapes: *node shapes* and *property shapes*. A node shape specifies constraints that must be met by the focus node itself, while a property shape focuses on the values of a particular property path from the focus node.

---

[1] https://www.w3.org/TR/shacl/

A typical shape can contain both node-level requirements and property-level constraints. For instance, one might say that all individuals of `ex:Person` must have a minimum of one `ex:ssn` value and that `ex:ssn` must not appear more than once per person. In SHACL, this rule can be written by embedding a *property shape* inside the node shape:

```
ex:PersonShape
  a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    a sh:PropertyShape ;
    sh:path ex:ssn ;
    sh:minCount 1 ;
    sh:maxCount 1
  ] .
```

Here, `sh:minCount` and `sh:maxCount` are examples of *cardinality constraints*, ensuring that each person node has at least one and at most one value for the property `ex:ssn`.

5.0.4 Core Constraints in SHACL

SHACL core constraints cover a wide variety of common data validation needs, including value type checking, range restrictions, and string pattern constraints. These may be applied individually or combined in more complex ways using logical operators. Some common constraints include:

- `sh:maxCount`, `sh:minCount` for specifying the cardinality of a property.

- `sh:maxLength`, `sh:minLength`, `sh:pattern` for string-based constraints.

- `sh:in` for enumerations, restricting property values to a given set.

- `sh:not`, `sh:and`, `sh:or`, `sh:xone` for expressing logical combinations of constraints.

Although these constraints can be listed succinctly, their actual usage in a shapes graph often involves more extensive descriptions, especially when specifying multiple shapes that share properties or when combining constraints via logical operators. An additional mechanism, the `sh:node` constraint, can require that property values themselves conform to another shape, facilitating nested validation.

5.0.5 Practical Observations and Example

Consider a small data graph:

```
ex:Alice  rdf:type   ex:Person .
ex:Alice  ex:ssn     "987-65-432A" .
ex:Bob    rdf:type   ex:Person .
ex:Bob    ex:ssn     "123-45-6789" .
ex:Bob    ex:ssn     "124-35-6789" .
ex:Calvin rdf:type   ex:Person .
ex:Calvin ex:birthDate "1971-07-07"^^xsd:date .
ex:Calvin ex:worksFor ex:UntypedCompany .
```

We might want to ensure that every person has exactly one `ex:ssn` and that those values match a certain pattern. The shape could be extended with a regular expression constraint, for instance:

```
ex:PersonShape
  a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    a sh:PropertyShape ;
    sh:path ex:ssn ;
    sh:maxCount 1 ;
    sh:pattern "^[0-9]{3}-[0-9]{2}-[0-9]{4}$"
  ] .
```

This enforces that the `ex:ssn` property can appear at most once and must match the typical US Social Security Number format. In a validation report, `ex:Alice` would fail the `sh:pattern` check, and `ex:Bob` would fail the `sh:maxCount` check because he has two distinct values for `ex:ssn`.

5.0.6 Improving RDF Quality with SHACL

By systematically applying constraints, SHACL helps identify common data issues, such as missing fields, invalid data formats, or undesired duplications. Organizations can thus use SHACL validation to ensure that their RDF datasets remain consistent with the intended domain model or schema guidelines. When violations occur, the SHACL engine produces detailed information about which shapes or constraints failed, allowing data curators to correct the problems swiftly. Consequently, leveraging SHACL promotes higher data integrity, making it especially valuable in production knowledge graphs where data is integrated from heterogeneous sources.

Moreover, recent community surveys on shape usage highlight that many practitioners create SHACL shapes manually, tailoring constraints to the specific modeling requirements of their domains. Automation tools do exist, but often struggle with data diversity and can produce overly general shapes, which then require considerable refining. As a result, an iterative approach of shape creation, testing, and adjustment is a common practice in real-world settings.

6 Knowledge Graph Creation

6.1 Introduction

Knowledge graphs integrate data from diverse sources into a coherent, semantically rich representation. Their fundamental building blocks are *entities* (resources) and *relationships* (edges) enriched with metadata, allowing machines and humans to derive context and meaning beyond the raw data itself. A typical challenge in knowledge graph creation is how to transform, map, and publish data—often stored in relational databases, spreadsheets, or JSON APIs—into RDF (Resource Description Framework), the primary data model for knowledge graphs.

Definitions of Knowledge Graphs. Knowledge graphs have been defined in a variety of ways. For instance:

- **Definition 1:** *"Knowledge graphs are large networks of entities, their semantic types, properties, and relationships between entities."*

- **Definition 2:** *"We define a Knowledge Graph as an RDF graph. An RDF graph consists of a set of RDF triples where each RDF triple* `(s, p, o)` *is an ordered set of the following RDF terms: a subject $s \in U \cup B$, a predicate $p \in U$, and an object $o \in U \cup B \cup L$. An RDF term is either a URI $u \in U$, a blank node $b \in B$, or a literal $l \in L$."*

  Here, U represents the set of URIs (Uniform Resource Identifiers) which uniquely identify resources, B represents the set of blank nodes used for anonymous resources, and L represents the set of literals which are values like strings, numbers, or dates.

One approach to building a knowledge graph is *template-based on-the-fly generation*, where existing web infrastructures dynamically render JSON, HTML, or other front-end representations from back-end databases. By using JSON-LD (JSON for Linked Data), it is possible to seamlessly embed structured semantic information into JSON documents. The added `@context` fields reference established vocabularies (such as `schema.org` or FOAF), aligning local properties with well-known semantic concepts. A browser or script not capable of interpreting RDF can still read the JSON as usual, while an RDF-aware processor or triple store extracts the semantics and stores them as triples. This makes JSON-LD particularly suitable for adding Linked Data to existing RESTful or web-based services without drastic changes in infrastructure.

A complementary strategy involves *mapping and transformation tools*. These tools convert data from formats like CSV, relational tables, or even unstructured text directly into RDF. For relational databases, the W3C-standard `R2RML` (RDB to RDF Mapping Language) provides a declarative mapping of database tables and columns to classes, properties, and individuals in an RDF graph. This workflow ensures a reproducible pipeline: one defines the mapping once, and the tool automatically transforms the data as needed, updating the knowledge graph whenever the source changes.

In practice, **organizations and developers often combine both template-based and mapping-based methods**. For instance, a web application could use on-the-fly JSON-LD generation for immediate API responses, while background processes or dedicated ETL (Extract–Transform–Load) jobs apply R2RML or related mappings to maintain a consolidated RDF store. Regardless of the chosen method, the underlying goal remains the same: to increase interoperability, enrich the data with domain semantics, and create a scalable infrastructure where diverse datasets come together as a comprehensive knowledge graph.

6.2   Tabular Data to RDF

When creating a knowledge graph, much of the source data often exists in tabular formats such as spreadsheets or CSV files. Traditional table-based data can be rich in information, but it typically lacks the semantic structure and linking conventions required for a knowledge graph. To bridge this gap, a popular solution involves using *OpenRefine*, a tool designed for cleaning, transforming, and reconciling messy data. The end goal is to export the refined data as RDF, thereby aligning it with a chosen vocabulary or ontology.

6.2.1   OpenRefine for Data Preparation

OpenRefine, originally developed by Google (under the name Google Refine), presents a user interface that resembles a spreadsheet application yet operates under a different philosophy. Rather than manually editing individual cells, OpenRefine applies transformations in bulk. This makes it easier to address common data wrangling tasks, such as:

- Splitting columns that contain composite values (e.g., multiple genres in one cell).

- Converting data types (e.g., converting string representations of dates into machine-readable date formats).

- Normalizing text (e.g., removing extra whitespace or punctuation).

While these tasks can be performed manually in small datasets, OpenRefine's real strength emerges when dealing with larger or more complicated tables. Its transformation steps are recorded in a reproducible script, enabling users to reapply them to different but structurally similar datasets. This *history of operations* can be shared with collaborators or reused on updated versions of the dataset, ensuring consistency in how the data is cleaned and prepared.

6.2.2   Entity Reconciliation

One of the most powerful features in OpenRefine is *entity reconciliation*, which provides a way to map textual identifiers or names in a table to canonical URIs in existing knowledge bases. For instance, you may have a column listing artist names, such as "The Beatles" or "Led Zeppelin." By leveraging an external reconciliation service (for example, Wikidata), OpenRefine attempts to match each artist name to an authoritative entity. If successful, it replaces the name in that column with a corresponding URI, providing a direct link to the external dataset. This step is crucial for producing data that is *linked* rather than merely enumerated, paving the way for more meaningful connections and queries within the knowledge graph.

6.2.3   Exporting to RDF

Once the data has been cleaned and entities resolved, the next challenge is to actually generate RDF from the refined table. For this, an OpenRefine extension named `RDF Refine` (previously developed by the Insight Centre in Galway and now maintained by the open-source community) offers a user-friendly interface to define how table columns map to RDF triples.

The extension uses a concept called *RDF skeletons* to specify the structure of the resulting triples. Each row in the table can be mapped to a resource in the RDF graph, and columns become predicates and objects according to the shape you define. As part of this skeleton, you can also integrate any external vocabulary by referencing namespaces or IRIs. For instance, a "Movie" could be linked to

a class `mv:Movie`, and attributes like `mv:hasName` or `mv:hasReleaseDate` might map to columns in the table.

When the skeleton is fully specified, OpenRefine can export the table to standard RDF formats such as Turtle or RDF/XML. In a typical example:

```
<http://example-instance.org/The%20Shawshank%20Redemption> a mv:Movie;
    mv:hasName "The Shawshank Redemption";
    mv:hasReleaseDate "1994-01-01T00:00Z";
    mv:hasGenre "Drama";
    mv:hasMovieDirector <http://example-instance.org/Frank%20Darabont> .
```

This snippet shows how each movie row becomes an instance of `mv:Movie`, with properties corresponding to data columns and a URI-based identifier. If an external reconciliation step was used, these URIs might instead link to resources already existing in larger knowledge graphs such as Wikidata or DBpedia.

6.2.4   Summary of steps

Using OpenRefine for tabular data to RDF conversion thus entails four main phases:

1. Importing and cleaning the table.

2. Performing bulk transformations (including date conversions or text splitting)

3. Reconciling entities with an external service

4. Defining an RDF skeleton to guide the final export

The resulting dataset can then be ingested into a triple store or combined with other RDF data to form a more complete knowledge graph.

For those looking to try out this approach, the official OpenRefine website offers installers and documentation, while repositories for the RDF Refine extension provide instructions on enabling RDF export features.

6.3   Relational Data to RDF

A considerable amount of structured data is stored in relational databases. Converting this information into RDF is a key step in creating knowledge graphs that interoperate with other Linked Data. The W3C's RDB2RDF recommendation outlines two main approaches for achieving this: the *Direct Mapping* and the more flexible *R2RML* (RDB to RDF Mapping Language). While Direct Mapping automatically exposes a relational schema in RDF, it typically offers limited control over the resulting vocabulary and structure. In contrast, R2RML empowers data engineers to fully customize how they wish to represent database rows, columns, and relationships in an RDF graph.

6.3.1   Direct Mapping Overview

Direct Mapping provides a baseline procedure for automatically generating triples from relational tables without requiring extensive configuration. Every table is mapped to a class in RDF, each row becomes a resource (with a generated URI), and columns are translated into predicates. This

approach can be useful for quickly publishing small databases or for experimentation, but it lacks explicit control over how to define the URIs, classes, or properties used in the final RDF representation. Consequently, the resulting graph may not use well-known vocabularies or only partially integrate with existing Linked Data.

6.3.2   R2RML: The RDB to RDF Mapping Language

R2RML offers the flexibility needed to create high-quality knowledge graphs. It is specified in RDF itself, meaning that one writes "mapping documents" describing how each table or SQL query, each row, and each column should be reflected in RDF. At a high level, an R2RML document consists of:

- `rr:logicalTable`: which table or query is being mapped,

- `rr:subjectMap`: instructions to generate the subject URI for each row,

- `rr:predicateObjectMap`: definitions of predicates and corresponding objects to produce the desired triples.

Instead of relying on generated names, developers can embed domain-specific vocabularies. For instance, a row in the "Person" table might be represented as an instance of `foaf:Person`, with properties such as `foaf:name` mapped from the `NAME` column. Within this framework, it becomes straightforward to:

1. Construct URIs from database identifiers, ensuring consistency and uniqueness.

2. Filter or transform data by using SQL queries instead of raw tables.

3. Reuse established ontologies and classes, thus integrating seamlessly with other Linked Datasets.

4. Interlink data by constructing URIs referencing external resources.

Below is a simplified example showing how a table named `Person` can be mapped into RDF resources of type `foaf:Person`:

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<TriplesMap1>
    a rr:TriplesMap ;
    rr:logicalTable [ rr:tableName "Person" ] ;
    rr:subjectMap [
        rr:template "http://www.ex.com/Person/{ID}" ;
        rr:class foaf:Person
    ] ;
    rr:predicateObjectMap [
        rr:predicate foaf:name ;
        rr:objectMap [ rr:column "NAME" ]
    ] .
```

In this snippet, each row in the `Person` table is turned into a `foaf:Person` whose URI is generated by replacing {ID} in the template with the value of the `ID` column. The `NAME` column is used as the object for the `foaf:name` predicate. Optionally, one can add further `rr:predicateObjectMap`

sections for additional columns (such as `birthdate`) or even embed more intricate logic to join multiple tables.

6.3.3 Advanced Configuration

For more complex use cases, R2RML supports:

- **SQL Views and Filters:** Instead of mapping entire tables, one can define sub-queries that filter or join. This ensures only the relevant data appear as triples.

- **Join Conditions:** Inter-table relationships can be expressed by specifying how a foreign key column in one table links to a primary key in another, resulting in proper RDF links between resources.

- **Term Maps:** Columns can become IRIs, blank nodes, or literals. This is especially powerful when certain textual values need to be interpreted as IRIs referencing external entities.

- **Data Type Casting:** Numeric or date columns can be cast to matching XSD data types, preserving semantic precision in the resulting RDF.

6.4 Structured Data to RDF

While relational databases are a common source for RDF conversion via R2RML, many real-world data sources are more varied: JSON documents, XML feeds, CSV files with irregular structures, and so on. The RDF Mapping Language (RML) was conceived as a **superset** of R2RML in order to deal with these heterogeneous formats in a uniform way.

6.4.1 RML Fundamentals and Motivation

RML extends the R2RML concept of a *logical table* (i.e., which part of the data one is mapping) by introducing the notion of *logical sources*. Instead of only referring to a table name or SQL query, an RML logical source can specify any input format and define how iteration over that data should occur. For example, JSON files often require a JSONPath expression to locate items, while XML might use XPath, and CSV might use a dedicated CSV-based query formulation. This generalization allows developers to maintain a consistent mapping approach across varied data formats.

The high-level structure of an RML mapping remains much the same as in R2RML:

- A `TriplesMap` block designates how certain portions of the input are transformed into RDF triples.

- A `rml:logicalSource` clause pinpoints the source and iteration method.

- A `rr:subjectMap` describes how to produce the subject URIs (or blank nodes) for each discovered record.

- One or more `rr:predicateObjectMap` sections define the RDF predicates and objects derived from fields in the source data.

However, the references in an RML mapping might point to JSON keys, XML elements, or CSV columns, depending on the `rml:referenceFormulation` in use. This mechanism gives a single specification language for data scattered among diverse file formats and structures.

6.4.2 RML in Practice

As a concrete illustration, one could have a JSON file named `people.json` containing:

```
{
  "persons": [
    { "firstname": "Alice", "surname": "Smith" },
    { "firstname": "Bob",   "surname": "Johnson", "knows": "Alice_Smith" }
  ]
}
```

An RML logical source might specify `rml:source "people.json"` and an iterator of `rml:iterator "$.persons[*]"`, denoting that each entry in the `persons` array represents a record to be processed. If the JSON items have fields like `firstname`, `surname`, and `knows`, the RML mapping can instruct the engine to construct a subject URI (e.g., `http://ex.com/Person/{firstname}_{surname}`) and assign `foaf:Person` as the class, while also generating `foaf:name` and `foaf:familyName` triples. Furthermore, it can treat the `knows` field as an IRI reference to another person's subject (e.g., linking `Bob` to `Alice`).

```
@prefix rr:   <http://www.w3.org/ns/r2rml#> .
@prefix rml:  <http://semweb.mmlab.be/ns/rml#> .
@prefix ql:   <http://semweb.mmlab.be/ns/ql#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .


<#PeopleJSON>
    rml:source "people.json" ;
    rml:referenceFormulation ql:JSONPath ;
    rml:iterator "$.persons[*]" .


<#PersonMap>
    rml:logicalSource <#PeopleJSON> ;
    rr:subjectMap [
        rr:template "http://ex.com/Person/{firstname}_{surname}" ;
        rr:class foaf:Person
    ] ;
    rr:predicateObjectMap [
        rr:predicate foaf:name ;
        rr:objectMap [ rml:reference "$.firstname" ]
    ] ;
    rr:predicateObjectMap [
        rr:predicate foaf:familyName ;
        rr:objectMap [ rml:reference "$.surname" ]
    ] ;
    rr:predicateObjectMap [
        rr:predicate foaf:knows ;
        rr:objectMap [
            rr:template "http://ex.com/Person/{knows}" ;
            rr:termType rr:IRI
        ]
```

```
    ] .
```

6.4.3 YARRRML: A More Accessible Syntax

For those who find Turtle-based RML definitions too cumbersome, a YAML-based syntax called **YARRRML** has emerged.
YARRRML scripts express the same core mapping instructions as RML but in a more lightweight, human-readable format. Developers can translate YARRRML documents into valid RML automatically and then execute them with standard RML engines.

Below is a YARRRML snippet defining a `personMap` that references the same `people.json` source and constructs URIs from the `firstname` and `surname` fields, while also linking `Bob` to `Alice` via `foaf:knows`:

```
prefixes:
  foaf: "http://xmlns.com/foaf/0.1/"
  ex:   "http://ex.com/Person/"

sources:
  peopleSource:
    access: "people.json"
    referenceFormulation: ql:JSONPath
    iterator: "$.persons[*]"

mappings:
  personMap:
    sources:
      - peopleSource
    subject:
      template: "http://ex.com/Person/{firstname}_{surname}"
      class: foaf:Person
    po:
      - predicate: foaf:name
        object:
          reference: "$.firstname"
      - predicate: foaf:familyName
        object:
          reference: "$.surname"
      - predicate: foaf:knows
        object:
          template: "http://ex.com/Person/{knows}"
          termType: IRI
```

This setup allows `Bob` to have an explicit reference to `Alice`, reflecting a small social graph in RDF. In real-world scenarios, additional mappings (or a separate look-up) could further resolve "Alice_Smith" to the unique subject URI of `Alice`, fostering richer interconnections in a knowledge graph.

6.5 Text to RDF

One of the most challenging aspects of knowledge graph creation is extracting structured information from free-text resources. While tabular and relational data typically follow well-defined schemas, text is unstructured by nature. Tools and frameworks for *semantic annotation* attempt to bridge this gap by detecting and linking concepts, entities, or relationships within text to RDF vocabularies and external Linked Data resources. This process can significantly enrich a knowledge graph, allowing it to capture both explicit and implicit information about entities mentioned in documents, articles, and other textual sources.

6.5.1 Semantic Annotation: Formal Modeling and Motivation

Semantic annotation refers to **attaching explicit, structured information (annotations) to segments of text**. In an RDF context, each annotation can be viewed as a statement that connects a piece of text (*the annotated data*) to a resource or concept (*the annotating data*) via a specific relation. Formally, an annotation A can be defined as a tuple (a_s, a_p, a_o, a_c), where:

- a_s is the subject being annotated (the text or document segment),

- a_p is the predicate expressing the nature of the annotation (e.g., `dbo:author` or `rdfs:comment`),

- a_o is the object (the concept or additional data that annotates a_s),

- a_c is contextual information, such as the provenance or scope of the annotation.

By representing these annotations in RDF, documents can be linked to external entities, making the information discoverable and machine-processable. For instance, a reference to "Stanley Kubrick" in a paragraph can be associated with the corresponding `dbr:Stanley_Kubrick` resource in DBpedia, thus connecting the text to a broad array of related data.

6.5.2 Annotation Ontologies and Standards

To promote interoperability and reusability of annotations, standard ontologies have emerged. One of the most prominent is the *Open Annotation Ontology*, now W3C's *Web Annotation Data Model*, which defines best practices for creating and sharing annotations across multiple platforms and domains. In this model:

- An `oa:Annotation` entity ties together the *Target* (the text segment or media being annotated) and the *Body* (the additional information or resource describing or contextualizing the target).

- A `SpecificResource` or a `Selector` can be used to pinpoint exact positions within a text, e.g., character offsets for a mention or a region in an image.

An example of the ontology can be seen in figure 10

6.5.3 Annotation Tools and Services

A variety of tools exist to perform semantic annotation, each bringing its own approach to Natural Language Processing (NLP), named entity recognition, and entity linking:
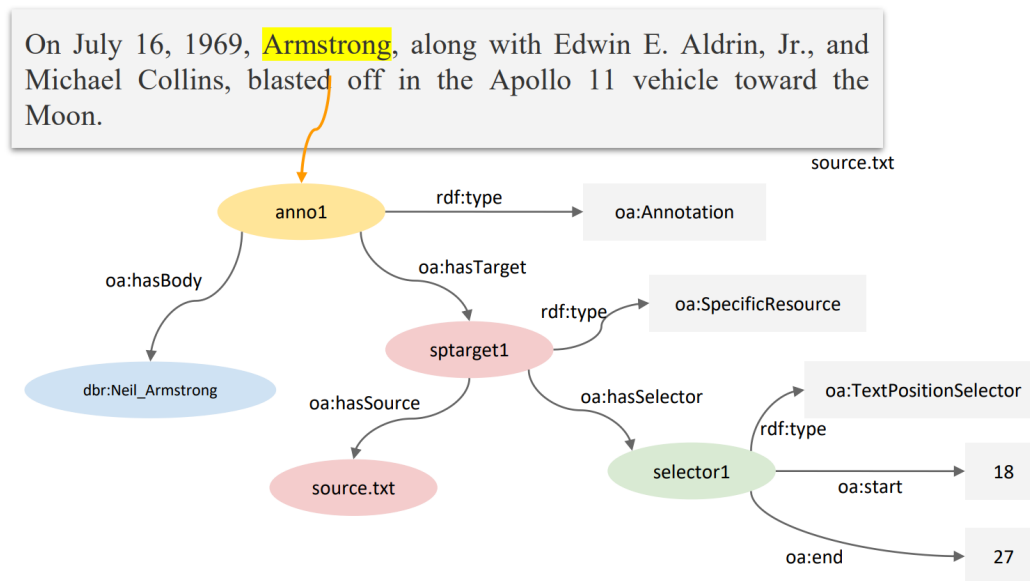
Figure 10: Example of semantic annotation

DBpedia Spotlight. This open-source tool detects mentions in text and links them to corresponding DBpedia URIs. It supports multiple languages and can be accessed via a web service, Java/Scala APIs, or plugins for languages like Python or PHP. When used in a pipeline, DBpedia Spotlight transforms unstructured text into a semantically richer form by exposing entities and properties that map back to the Linked Open Data (LOD) cloud.

PermID (OpenCalais). Offered by Refinitiv (formerly Thomson Reuters), the PermID provides powerful text analytics for identifying entities, relationships, and events. Though it is a commercial product, it integrates well with existing knowledge graphs via stable PermIDs, ensuring consistent entity URIs across different data sources.

GATE (General Architecture for Text Engineering). An established open-source framework for text analysis, GATE provides a configurable environment for building *bespoke* annotation pipelines. Developers can combine pre-built NLP components—tokenizers, part-of-speech taggers, named entity recognizers—with domain-specific modules to tailor extraction for specialized tasks. This is especially helpful when the text is highly domain-specific (medical, legal, etc.) and general-purpose annotators produce suboptimal results.

6.5.4 From Document to Knowledge Graph

Semantic annotation is most beneficial when the resulting annotations are expressed in RDF and integrated into the broader knowledge graph. A typical workflow might include:

1. **Text Preprocessing.** Cleaning the text, splitting it into sentences or sections, and normalizing characters.

2. **Annotation.** Passing each document to a chosen tool (e.g., DBpedia Spotlight or GATE) that detects entity mentions, relationships, or other relevant features.

3. **Triple Generation.** Converting discovered annotations into RDF triples, referencing vocabularies like `dbo:`, `foaf:`, `schema:`, or application-specific predicates. The text segments themselves may become subjects (or remain as literal strings) in the final RDF model.

4. **Storage and Query.** Persisting the generated triples in a triplestore or knowledge graph. Queries can then combine structured data, such as *which entity occurs in which document segment,* or advanced questions that blend text-derived knowledge with relational or hierarchical information from other parts of the graph.

By adopting standard models and robust annotation pipelines, text can be transformed from inert strings into linked resources. This provides deeper insights, fosters content discovery, and facilitates data-driven analysis. In large organizations, such a process enables knowledge workers to locate all references to a particular entity across numerous documents, bridging the gap between unstructured text and structured data management.

6.6 RDF Storage

Storing RDF data efficiently is an important step in building knowledge graphs that scale to large or complex datasets. A core challenge lies in balancing the flexibility of the triple-based data model with performance concerns typically associated with high-volume query processing. Numerous storage strategies have emerged over the years, each making different trade-offs regarding indexing, data layout, and distribution. Broadly, we can divide these storage architectures into *centralized* and *distributed* systems, depending on whether they run on a single node or across a cluster.

6.6.1 Centralized vs. Distributed Systems

A *centralized* RDF store runs on a single node, managing all data and queries locally. These systems tend to be simpler to configure and can be highly optimized for small- to medium-scale datasets. In contrast, *distributed* systems partition data across multiple nodes. This approach accommodates very large datasets and heavy query loads, at the expense of increased complexity in data partitioning and query coordination. As data volume grows and query workloads intensify, distributed solutions may become necessary to maintain acceptable performance.

6.6.2 Common RDF Storage Approaches

Although designs vary, several canonical approaches dominate the landscape of RDF storage. Each approach offers distinct strengths depending on the nature of the data and the query patterns expected:

Statement Tables. Often considered the most direct strategy, *statement tables* store each triple as a row in a universal table with columns for subject, predicate, and object. This linearized structure requires minimal preprocessing and naturally supports the triple model. However, queries involving multiple predicates often necessitate expensive self-joins, reducing performance when the dataset or query complexity grows. Systems such as Jena SDB, 3Store, 4Store, and Virtuoso can store data in statement-table-like formats, sometimes with additional indexing to mitigate join costs.

Property Tables. The *property table* approach attempts to reduce self-joins by placing commonly co-occurring predicates (properties) into a single table with multiple columns. Each row in this table

corresponds to a subject, with columns for each property. Clustering these related properties can reduce the number of joins needed for queries that frequently access them together. Systems like DB2RDF and Jena2 SDB adopt property tables in certain configurations, although deciding which properties to cluster is itself a nontrivial design issue.

Index Permutations. An alternative strategy is to build multiple indexes of the triple data in different permutations of subject, predicate, and object. Systems such as *Hexastore* and *RDF-3X* index all six permutations of the triple components (SPO, SOP, PSO, POS, OSP, OPS). By providing every possible ordering as an index, queries can look up triples efficiently, often leading to very fast merge joins. The trade-off lies in increased storage overhead from maintaining so many indexes, as well as the update cost when data changes.

Vertical Partitioning. A *vertical partitioning* approach creates one table per property in the dataset. Each table contains two columns: subject and object. Multi-valued properties lead to additional rows within the same property table, and each table is usually indexed by subject. Because queries that filter by property can scan only the relevant table, the approach often leads to faster retrieval for property-specific queries. However, queries involving many properties simultaneously may require multiple lookups, potentially complicating join logic. The *SW-Store* system is often cited as a prominent example of vertical partitioning.

6.6.3   Selecting the Right Approach

Choosing an appropriate RDF storage strategy hinges on the expected usage patterns, dataset size, and operational constraints:

- **Frequent multi-property queries:** Consider property tables or index permutation systems. By bringing together related data or offering multiple access paths, they reduce the number of expensive joins.

- **Extremely large or distributed datasets:** Evaluate distributed systems that partition data and parallelize queries. Vertical partitioning or statement tables distributed across nodes can also help, although the orchestration layer becomes more involved.

- **Simplicity and generality:** A straightforward statement-table design can be implemented quickly and is flexible enough to handle arbitrary RDF, making it suitable for prototyping or modest dataset sizes.

Overall, there is no one-size-fits-all solution for RDF storage. Instead, engineers typically test multiple approaches, balancing storage overhead and query performance while considering ease of implementation. Understanding these core techniques provides a solid foundation for deploying an RDF backend that can support knowledge graph creation and querying at scale.

7 Extras

The evolution of knowledge graphs—spanning research endeavors and widespread industry adoption—demonstrates their power to integrate, manage, and extract value from vast and heterogeneous datasets. Over the years, we have witnessed significant breakthroughs, from the first conceptualizations of the Semantic Web to today's AI-driven initiatives that fuse symbolic reasoning with deep learning. Below is a concise look at emerging themes and applications that illustrate how knowledge graphs continue to transform data management and analytics across various domains.

7.0.1 Historical Context and Transition to Knowledge Graphs

The foundational ideas of semantic systems and knowledge representation date back to Tim Berners-Lee's vision of the Semantic Web, famously articulated in the 2001 *Scientific American* article on the future of web data. The ultimate "killer app" envisioned was an ecosystem of intelligent agents capable of automatically scheduling appointments, negotiating services, and making decisions on behalf of users, all by leveraging standardized data representations. While this level of agent autonomy remains a long-term goal, the building blocks—structured data, ontologies, and Linked Data principles—have materialized in a more immediate sense through knowledge graphs that power modern search engines, recommender systems, and advanced analytics.

7.0.2 Industry Impact and Linked Data Growth

Industry adoption of knowledge graphs has accelerated in the past decade. Large-scale enterprises (e.g., in the web search, e-commerce, and social media sectors) leverage graphs to store and query factual data about billions of entities, vastly improving the quality and intelligibility of their products. Meanwhile, open knowledge graphs such as DBpedia, Wikidata, Bio2RDF, and the Linked Open Data cloud have continued to expand, providing a wealth of structured data that benefits various academic and commercial applications. This openness fosters collaborative cross-domain analysis, enabling novel insights in areas from biomedical research to climate monitoring.

7.0.3 Neurosymbolic AI

A notable trend is the rise of *neurosymbolic AI*, which aims to unify neural networks (pure machine learning) with symbolic knowledge reasoning (pure logic). By combining robust pattern recognition with the interpretability and structure of symbolic reasoning, we can craft AI systems that:

- Use knowledge graph edges and types to guide or constrain neural models.

- Provide traceable, explainable outcomes, helping domain experts scrutinize how decisions are reached.

- Handle incomplete or uncertain data in a consistent framework, often relying on partial completeness assumptions or uncertain reasoning.

The knowledge graph thus acts as the backbone, offering an interpretable semantic layer for data-driven algorithms.

7.0.4   Data Fabrics and Semantic Data Lakes

Traditional data warehousing models are often being superseded or supplemented by *data lakes*—unified repositories of raw data from diverse sources. However, without semantic annotation or modeling, these lakes quickly become "data swamps." *Data fabrics* and *semantic data lakes* introduce knowledge graphs as a unifying schema layer, ensuring consistent interpretations of data elements across organizational silos. By employing standard ontologies and cross-dataset links, data fabrics can power advanced analytics, machine learning, and real-time data integration. This approach is crucial in contexts like scientific data management (e.g., integrating genome sequences with soil metadata and satellite imagery) or large-scale enterprise data catalogs.

7.0.5   Provenance and Explainability

As systems become more data-driven, there is heightened demand for *explainability* and *provenance*. Users increasingly want to know:

- *Where* a piece of information originated.

- *How* a query result was derived (which triples or inference rules contributed to the final answer).

- *Why* certain data was included or excluded in an analysis.

Research in SPARQL *how-provenance* addresses these questions by assigning identifiers to triples and rewriting queries to track contributions to each result. In parallel, frameworks like PROV-O define ontologies for describing the provenance of data transformations. Together, these solutions bolster trust and interpretability in knowledge graphs used for mission-critical tasks.

7.0.6   Use Cases and Future Directions

Healthcare and Bioinformatics.   From electronic health records to genomic data, knowledge graphs can unify patient information, medication histories, and disease ontologies. Combined with explainable AI techniques, clinicians can better trace diagnosis steps or medication recommendations.

Sustainability and Circular Economy.   Governments, organizations, and researchers employ knowledge graphs for environmental data integration—ranging from complex climate models to local land-use data. Semantic annotation of policy documents and green finance data fosters transparency and shapes evidence-based decision-making.

Media, Entertainment, and Cultural Heritage.   By semantically annotating media assets and historical archives, cultural institutions can create rich interactive experiences, enabling new forms of curation and cross-collection linkage.

Transportation and Logistics.   Large-scale logistics operations rely on integrated data about routes, cargo, and weather conditions. Knowledge graphs, combined with federated queries, can unify these streams and support real-time decision-making.

Manufacturing and Industry 4.0.   Smart factories adopt semantic layers to model machine components, processes, and IoT sensor data. The resulting knowledge graphs improve process optimization, fault detection, and supply chain coordination.

Looking forward, knowledge graphs will continue to expand their horizons by blending advanced reasoning, neural models, and streaming data sources. These developments will call for scalable storage, query optimization, and robust data governance. It is exactly in this interplay of technology, data management, and semantics that knowledge graphs shine—turning raw data into contextualized, intelligent, and actionable knowledge.

8 Questions from previous Exams

8.1 True or False Questions

correct +1
wrong -1
no answer =0

8.1.1 Exam - 19. January 2024

1. RDFS allows the definition of subproperties
   TRUE, rdfs:subPropertyOf, defines property hierarchies where related resources by a subproperty are also related by its superproperty

2. SHACL is equivalent to OWL.
   FALSE, OWL extends RDF and RDFs and is used to represent ontologies, SHACL is designed for validating and describing constraints on RDF graphs.

3. Mathematically, a knowledge graph can be described as a directed labelled graph.
   TRUE, it can be described as a directed labeled Graph Subject → Predicate → Object.

4. Blank nodes can occur in any position of an RDF triple.
   FALSE, Blank node identifiers may only appear as subjects or objects in triples (not predicates).

5. RML allows to specify XML sources.
   TRUE, rml:referenceFormulation can be used to specify XML sources using XPath.

6. N-Triples are a valid serialization format of RDF triples.
   TRUE, each RDF Statement (triple) is terminated by period (".").

   ```
   <http://musicbrainz.org/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>
   <http://xmlns.com/foaf/0.1/name>
   "The Beatles".
   <http://musicbrainz.org/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d#_>
   <http://www.w3.org/2002/07/owl#sameAs>
   <http://dbpedia.org/resource/The_Beatles>.
   ```

   In this example:
   The subject is http://musicbrainz.org/artist/b10bbbfc-cf9e-42e0-be17-e2c3e1d2600d# .
   The predicate (property) is either http://xmlns.com/foaf/0.1/name¿ or ¡http://www.w3.org/2002/07/ow
   The object is "The Beatles" (a literal) or http://dbpedia.org/resource/The Beatles (a URI).

7. Evaluating a SPARQL query with entailment regimes always leads to fewer results.
   FALSE, You get more results thanks to the reasoning engine.

8. "?" is used to denote blank nodes in RDF.
   FALSE, variables aka. "?" are used to describe placeholders in SPARQL Queries.

9. OWL ontologies can be exchanged as RDF documents.
   TRUE, OWL ontologies are typically expressed in RDF syntax, enabling easy integration and sharing on the Semantic Web.

10. The "Open World Assumption" allows us to consider missing information as negative information.
    FALSE Missing information is NOT treated as negative information. **Closed World Assumption** treats missing data as FALSE.

11. Using RDFS, we can express the fact that every child has exactly one father.
    FALSE, RDFS cannot be used for that, OWL extends RDFS by using cardinality and other properties which enable this expression in OWL.

12. RDFS allows multiple inheritance for classes and properties.
    TRUE, by using e.g. rdfs:subPropertyOf multiple inheritance is possible.

13. The result type of an ASK query is an RDF graph or triple.
    FALSE, ASK returns a boolean indicating wetherany solution exists.

14. SPARQL uses primary keys for subgraph merging.
    FALSE, it uses URIs to identify resources uniquely.

15. The result type of a CONSTRUCT query is a Boolean.
    FALSE, it returns the new and updated RDF graph.

16. Internationalized Resource Identifiers (IRIs) are a subset of Uniform Resource Identifiers(URIs).
    FALSE Internationalized Resource Identifiers (IRI) is an extension of Uniform Resource Identifier(URI).

17. A SPARQL query can be evaluated on the default graph.
    TRUE, if no named graph is specified, SPARQL operates on the default graph.

18. R2RML is a superset of RML.
    FALSE RML is a superset of R2RML.

19. SPARQL supports JOIN clauses.
    FALSE There is no extra "join" clause, joins are performed implicitly whenever multiple triple patterns are included in the WHERE clause.
    Example:

```
SELECT ?person ?name ?email
WHERE {
  ?person foaf:name ?name .
  ?person foaf:mbox ?email .
}
```

20. RDFS provides a wider range of ontological constructs than OWL.
    FALSE, Owl extends RDF and RDFS and therefore has a wider range. It enables more expressive and complex descriptions of knowledge.

8.1.2   Exam - 15. March 2024

1. On the subject position of an RDF triple, any RDF term is allowed.
   FALSE, Literals are not allowed, URIs and Blank nodes are allowed.

2. In SPARQL queries, SELECT DISTINCT filters out duplicate solution mappings.
   TRUE, SELECT DISTINCT removes duplicate entries.

3. ASK queries return an RDF graph.
   FALSE, ASK queries return a BOOLEAN.

4. SPARQL queries are always translated into SQL and then executed on a relational database.
   FALSE, they are used for RDF data and directly executed on RDF stores.

5. In a SPARQL query, only solutions that match all of the FILTER expressions are returned.
   TRUE, all of the FILTER expressions need to return TRUE to be included in the solution.

6. A resource can only be a member (hint: type) of a single class.
   FALSE, a resource can be a member of multiple classes.

7. Blank nodes have URIs.
   FALSE, Blank nodes have no URI.

8. Domain and range of properties can be specified in RDFS.
   TRUE, rdfs:range, rdfs:domain can be used for that.

9. RDFS is more expressive than OWL.
   FALSE, OWL extends RDFS and therefore is more expressive than RDFS.

10. In OWL, the range of object properties are instances of classes.
    TRUE, the range of an object property specifies the class to which the objects of the property must belong.

11. The result type of a CONSTRUCT query is an RDF graph or triple.
    TRUE, a CONSTRUCT query returns the updated RDF graph.

12. SPARQL is primarily a query language for relational databases.
    FALSE, it is a query language for RDF graphs.

13. The result type of an ASK query is a Boolean.
    TRUE

14. Evaluating a SPARQL query with entailment regimes can result in more results.
    TRUE, through inference e.g. entailment regimes (RDFS or OWL-based reasoning) the query result can increase.

15. A SPARQL query can only be evaluated on the default graph.
    FALSE, it used the default graph by default but can also query named graphs.

16. SHACL is mainly targeted at XML.
    FALSE it is primarily designed for validating and defining constraints on RDF graphs.

17. RML is a superset of R2RML.
    TRUE, RML is a superset of R2RML, RML is an extended version of R2RML to handle different data formats in a consistent way.

18. R2RML allows to specify XML sources.
    FALSE, RML allows to specify XML sources.

19. RDFS is a way to specify Ontologies.
    TRUE, just like RDF and Owl it is used to specify Ontologies and build graph based models.

20. OWL is based on a form of Logic.
    TRUE

8.2   Open Questions, 4 Points each

8.2.1   Exam - 19. January 2024

1. Write a query that checks whether thriple (:s, :p, :o) occurs in RDF graph http://example.org/g.ttl

```
ASK {
  GRAPH <http://example.org/g.ttl> {
    :s :p :o .
  }
}
```

A triple is a basic unit in RDF that represents a fact. It has three parts: :s (subject): The thing we're talking about.
:p (predicate): The property or relationship of the subject.
:o (object): The value or thing connected to the subject.
The ASK query returns TRUE or FALSE depending on if this triple exists. The GRAPH defines which Graph we want to check.

2. Given the following query:

```
SELECT ?x ?z
WHERE {
{ ?x :p ?z . }
UNION
{ ?x :q _:y . _:y :q ?z . }
}
```

Identify the triple patterns and the basic graph patterns in the query.
A triple pattern is a statement in the query that resembles an RDF triple but may contain variables (e.g., ?x, ?z, _:y) instead of fixed values.

- The first block (?x :p ?z .) contains 1 triple pattern: ?x :p ?z

- The second block (`{ ?x :q _:y . _:y :q ?z . }`) contains **2 triple patterns**: ?x :q _:y _:y :q ?z

A basic graph pattern is a set of one or more triple patterns grouped together within a single query block.

- BGP 1:   ?x :p ?z .   (a single triple pattern).

- BGP 2:   ?x :q _:y . _:y :q ?z .   two triple patterns combined in one block.

3. Consider the following OWL statements:

```
:Writer a owl:Class .
:Author a owl: Class .
```

```
:Poet a owl:Class ;
rdfs: subClassof :Writer .
:Writer a owl: Class ;
rdfs: subClassOf :Creative .
:Writer owl:equivalentClass :Author .
```

List at least two OWL statements (not already listed above) that are entailed by the above.
:Poet rdfs:subClassOf :Creative .
Explanation: :Poet is a subclass of :Writer and :Writer is a subclass of :Creative
:Author owl:equivalentClass :Writer .
Explanation: :Writer is equivalent to :Author and this is symmetric.

4. Please formulate the following information in RDF by making a statement about a statement.

   ```
   The statement "McCoy is a doctor" is attributed to Kirk.
   ```

   Solution:

   ```
   ex:Kirk :says _:s1 .
   _s1 a rdf:Statement;
       rdf:subject ex:McCoy;
       rdf:predicate ex:isA;
       rdf:object ex:Doctor.
   ```

   Or in RDF*

   ```
   <<ex:McCoy ex:isA ex:Doctor>> :says ex:Kirk
   ```

8.2.2   Exam - 15. March 2024

Assume that based on the data provided on DBpedia, you want to retrieve information about music. Consider the following example data from DBpedia about "Taylor Swift", This is just a small sample of all the data about music to be found in DBedia. However, the example provides an insight on how the data is modeled and represented.

```
@prefix dbp: <http://dbpedia. org/property/>
@prefix dbr: <http://dbpedia.org/resource/>
@prefix dbo: <http://dbpedia.org/ontology/>
@prefix dbd: <http://dbpedia.org/datatype/>
@prefix rdf: <http://www.w3.0rg/1999/02/22-rdf-syntax-ns#
@prefix schema: http://schena.org/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
dbr: Taylor_Swift
rdfs:label
"Taylor Swift"@de ,
"Taylor Swift »@es ,
"Taylor Swift"@pl,
"Taylor Swift »@pt
rdf: type
```

```
dbo: Artist,
dbo:MusicalArtist
dbp: genre
dbr: Pop_music ;
dbr:Country_music ;
dbp: instrument
"piano"@en ,
"banjo"@en ,
"ukulele"@en ,
"guitar"@en ,
"Vocals"@en ;
dbp: yearsActive
2004 ;
dbo: birthDate
« 1989-12-13 »^^xsd:date ;
foaf: name
"Taylor Swift"@en ;
foaf: homepage
<http://taylorswift.com>
dbr: 1989_ (Taylor Swift_album)
dbp: artist
dbr:Taylor_Switt i
foaf: name
1989"@en ;
rdf: type
schema: MusicAlbum ,
dbo: Album;
dbp: released
"2014-10-27"^^xsd: date ;
dbp: salesamount
"5124000"^^xsd:integer
dbr: Blank_Space
dbo: album
dbr: 1989_ (Taylor_Swift_albun) ;
dbp: writer
dbr: Max_Martin ;
foaf: name
"Blank Space"@en;
dbo: runtime
231.0 . #Runtime in second
```

Assume that all data on music is stored in the Default Graph in the SPARQL engine.
Further assume that the prefixes that are used in the graph are also defined for the SPARQL
engine, i.e., start your solution directly with "SELECT ... ".
Write SPARQL queries for the following tasks.

1. Return the IRIs of all pop artists. (Hint: genre)

```
SELECT DISTINCT ?artist
WHERE {
  ?artist dbp:genre dbr:Pop_music .
  ?artist rdf:type dbo:Artist .
}
```

2. Return the names of albums and their release dates. (Hint: album)

```
SELECT DISTINCT ?albumName ?releaseDate
WHERE {
  ?album rdf:type dbo:Album ;
         foaf:name ?albumName ;
         dbp:released ?releaseDate .
}
```

3. Given the following shape graph: Please provide an example data graph, i.e., a set of RDF triples, where (i) at least one entity/ resource/node is valid with respect to the given constraints and (ii) at least one entity/resource/node violates the highlighted constraint but fulfills the others specified in the shape graph.

```
ex:PersonShape rdf:type sh:NodeShape
ex:PersonShape sh:targetclass ex:Person .
ex:PersonShape sh:property [
rdf:type sh:PropertyShape ;
sh:path ex:ssn ;
sh:maxCount 1 ].
```

Solution:

```
ex:Alice rdf:type ex:Person ;
         ex:ssn "123-45-6789" .


ex:Bob rdf:type ex:Person ;
       ex:ssn "987-65-4321" ;
       ex:ssn "555-55-5555" .
```

Explanation:
Valid entity: Alice satisfies the constraints.
Invalid entity: Bob violates the sh:maxCount constraint

4. Consider the following OWL statements,

```
:Film a owl:Class .
:Movie a owl:Class
:ActionMovie a owl:Class ;
rdfs:subClassOf :Film .
:Film a owl:Class ;
rdfs:subclassof :Creativework .
:Film owl:equivalentclass :Movie.
```

List at least two OWL statements (not already listed above) that are entailed by the above.
Solution:

```
:ActionMovie rdfs:subClassOf :CreativeWork .
:Movie rdfs:subClassOf :CreativeWork .
```

5. Consider the follwoing R2RML mapping:

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<TriplesMap1>
    a rr:TriplesMap;
    rr:logicalTable [ rr:tableName "Person" ];
    rr:subjectMap [
        rr:template "http://www.ex.com/Person/{ID}";
        rr:class foaf:Person
    ];
    rr:predicateObjectMap [
        rr:predicate foaf:name;
        rr:objectMap [ rr:column "NAME" ]
    ].
```

And the following table:

ID	NAME	DEP	REF
10	Claire	A	A1
20	Dave	B	A1

Table 2: Person Table

Solution: The output of the mapping in Turtle syntax:

```
@prefix ex: <http://www.ex.com/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:Person/10 a foaf:Person ;
             foaf:name "Claire" .

ex:Person/20 a foaf:Person ;
             foaf:name "Dave" .
```